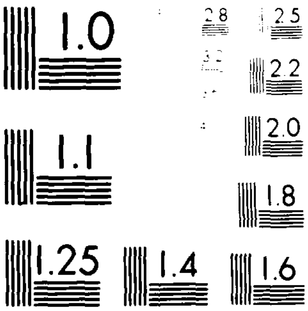
```
AD-A111 981   OHIO STATE UNIV  COLUMBUS COMPUTER AND INFORMATION SC--ETC  F/G 9/2
              THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS). P--ETC(U)
              JAN 82  D S KERR, A OROOJI, Z SHI                   N00014-75-C-0573
UNCLASSIFIED  OSU-CISRC-TR-82-1                                             NL
```

1 of 2
AD A
1 981



MICROCOPY RESOLUTION TEST CHART

[illegible]

TECHNICAL REPORT SERIES

12

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY   COLUMBUS, OHIO

(OSU-CISRC-TR-82-1) 12

THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS): PART I - SOFTWARE ENGINEERING STRATEGIES AND EFFORTS TOWARDS A PROTOTYPE MDBS

by

Douglas S. Kerr
Ali Orooji
Zong-Zhi Shi
Paula R. Strawser

Work performed under
Cont act N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, OH 43210
January 1982

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OSU-CISRC-TR-82-1	2. GOVT ACCESSION NO. AD-A111 9	3. RECIPIENT'S CATALOG NUMBER 84
4. TITLE (and Subtitle) "The Implementation of a Multi-Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS"		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Douglas S. Kerr Zong-Zhi Shi Ali Orooji Paula R. Strawser		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, Virginia 22217		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE January 1982
		13. NUMBER OF PAGES 156
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR,. Boston
NRL 2627	ONR, Chicago
ONR 1021P	ONR, Pasadena

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

backend database system, database system implementation, [illegible]ase computer, database machine, software engineering, database.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A backend multi-minicomputer database system, known as MDBS, has been proposed. MDBS utilizes one minicomputer as the master (or controller) and a varying number of minicomputers as slaves (or backends) which are configured in a novel and parallel fashion. MDBS is primarily designed to provide for database growth and performance enhancement by the addition of identical backends. The software architecture allows the backend addition without the need of new programming and reprogramming. Instead, the backend system software is replicated on the new backends for concurrent and parallel operations which in turn allow the database to →

grow and the performance to improve without an increase in software complexity.

Prototypes of MDBS are being implemented in order to carry out design verification and performance evaluation of MDBS. The types of design verification and performance evaluation of MDBS to be conducted are discussed in the report. The prototypes will be developed in versions starting with a very simple version, i.e., MDBS-I, that is described in detail in this report. Four more versions are envisioned. The rationale for each of the subsequent versions is also given.

As the first in a series of reports on the implementation, this report discusses the choice of hardware and operating system software. It also discusses the choice of the system programming language.

The project is being used as an experiment in implementation methodologies and software engineering techniques. Thus, the report discusses the methodologies and techniques used, including a modified chief-programmer-team organization, structured walkthrough, data and service abstractions, a formal systems design language, and structured coding. The choice of a 'black-box' testing strategy is also discussed.

The MDBS-I software system architecture is described in some detail. In particular, the portion of the system which processes the information about the database, i.e., the directory data, is described. In order to use a database that already exists, a subsystem to convert and load the database will be provided. The database load subsystem is therefore described. Finally, in order to facilitate performance evaluation experiments, a program to generate test data is provided.

The final section of the report provides a preliminary discussion of alternative approaches for the operating system interface. Both a message-oriented approach and a procedure-oriented approach are examined for the purpose of supporting concurrency control of MDBS which is to be incorporated in the second version of MDBS, i.e., MDBS-II.

The appendices contain the detailed designs for the directory management subsystem, the database load subsystem and the test data generation program. Later reports will describe subsequent versions of the multi-backend database system, namely, MDBS-II, MDBS-III, MDBS-IV and MDBS-V.

TABLE OF CONTENTS

LIST OF FIGURES

PREFACE

We would like to thank all those who have helped with this project. In particular, the whole effort is based on the design and analysis developed by Jai Menon. (Now, Dr. Jai Menon of IBM Research Laboratory, San Jose.) He also provided much help in the detailed designs that we have documented herein. Tamer Ozsu was involved in our initial design efforts, before he left the project. Two Visiting Scholars, Xin-Gui He and Masanobu Higashida, who recently joined our project, helped with the program testing. Three undergraduates provided programming support: Steve Barth, Julie Bendig and Ray Browder. Several graduate students contributed background material which was necessary for making our design decisions: José Alegria, Tom Bodnovich, and David Braun. Finally, we would like to thank Prof. David K. Hsiao for the leadership and inspiration he has provided to our project.

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Professor of Computer and Information Science, and conducted at the Laboratory for Database Systems Research of the Ohio State University. The Laboratory for Database Systems Research of the Ohio State University is funded by the Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and the Ohio State University (OSU) and consists of the staff, graduate students, undergraduate students, Visiting Scholars and faculty for conducting research in database systems. Dr. Douglas S. Kerr, Associate Professor of Computer and Information Science is the Director of Laboratory. The research contract was administered and monitored by the Ohio State University Research Foundation. This report is issued by the Computer and Information Science Research Center (CISRC).

1.0 AN INTRODUCTION TO HIGH-PERFORMANCE AND GREAT-CAPACITY DATABASE SYSTEMS

Traditionally, database management systems run as large software packages (e.g., TOTAL) on large host computers (e.g., IBM 3033). Such systems have had problems with performance; i.e., as the database grows and the rate of requests to the database system increases, the host computer performance decreases. Instead of upgrading the host to a more powerful and expensive model (say, IBM 3081) and incurring a major system interruption, it has been proposed [Cana74] to offload most of the database system software from the host to a second computer system, known as the backend, thus freeing the existing host computer for other tasks.

One backend approach is to use a single minicomputer for the backend. This approach can free up the host, thereby improving the system performance for other tasks. However, if the database continues to grow and the rate of requests continues to increase, this approach cannot solve the database performance problem since the backend will soon be overloaded. Consequently, its pertormance will be degraded just as the host's would have been in the traditional approach. Thus, overall performance of the host and backend will be degraded. This approach is known as the single software backend approach.

A second approach to solving the database system performance problem is to develop a special-purpose database machine with specially designed hardware. However, the cost-effectiveness of this approach, known as the hardware backend approach, has not yet been demonstrated.

A third approach is to use multiple mini-computers configured in a novel and parallel way for performance improvement in order to allow for database growth and for increases in the request rate. This approach also requires the development of an innovative software design which allows the addition of mini-computers of the same type and the replication of the software on the new mini-computers without major system interruptions. Thus, it does not require the development of any new hardware, but only the development of a new and replicatable software architecture and a new and parallel hardware configuration. Because it allows the use of multiple mini-computers, this approach will result in a multi-backend database system.

In this report we describe the current status of the development of a prototype of such a multi-backend database system known as MDBS. By a prototype we mean one that has enough of the functionality of the full system to allow meaningful experiments to be conducted. However, some features that would be essential for a full system would be omitted in order to simplify the current implementation effort. The functionality provided and features omitted will be described in later sections.

1.1 Multi-Backend Database System Design Goals

The major goals we are trying to achieve are to design a multi-backend database system that allows the database to grow and the rate of requests to increase while maintaining good overall performance. In particular, a "good" multi-backend database system with high performance and great capacity should have the following properties:

(1) Throughput improvement is proportional to the number of backends. In other words, if the number of backends and disk drives is doubled, it should be possible to nearly double the size of the database and to nearly double the request rate on the database system.

(2) Response time is inversely proportional to the number of backends. It should also be possible to nearly halve the average response time by doubling the number of backends.

(3) System is extensible for capacity growth and/or performance improvement. By extensibility of a multi-backend database system we mean that upgrade of the system can be provided with no modification to existing software and no new programming; no modification to existing hardware; and no major disruption of system activity when additional hardware is being incorporated into the existing hardware and software.

This kind of extensibility is to be provided by designing a system with one controller (i.e., the master mini-computer) and several backends (slave mini-computers) where the design allows expansion by the addition of more backends of the same type, instead of by the replacement of the present backends with more powerful and expensive models. It also allows identical software to run on each backend, including new backends added for expansion.

This kind of extensibility calls for a design which minimizes the role of the controller of the backends so that it will not become a bottleneck after the addition of only a few backends.

1.1.1 Design Issues

Three types of design issues are addressed: hardware issues, system issues, and software issues. They are discussed in detail in [Hsia81a] and [Hsia81b]. Here we will review the hardware and system issues and solutions briefly. Since the software issues are closely related to our implementation effort, we will discuss their solutions more elaborately in the next section. In this chapter, definitions will be kept informal. More precise definitions can be found in Chapter 3 or in the previously mentioned reports, i.e., [Hsia81a] and [Hsia81b].

The hardware issues include the problem of backend interconnection -- Should the backends communicate with each other via some kind of interconnection hardware? How can this interconnection be provided in a cost-effective manner? The hardware issues also include the problem of database store interconnection -- Should each disk be accessible by all the backends, by only one backend, or by some but not all the backends?

Several system issues are addressed: Database placement -- Should related records of a database be concentrated at one backend or should they be distributed across several backends? If the records are to be distributed across several backends, how should this distribution be done? Execution mode -- should all backends process the same request in parallel or should different backends process different requests concurrently? Directory structure, placement, and management -- How should the auxiliary information about the database be determined, organized and distributed among the backends? Access control capability -- What are the kind and granularity of the access control and how should an access control mechanism be implemented? Data model and manipulation language -- what data model should be supported and what data manipulation language should be used?

The software issues include the problem of degree of concurrency -- Since the basic unit for processing is a request, should two or more requests be processed concurrently? If requests are not processed concurrently, what should be done when users submit groups (i.e., transactions) of requests? Can the processing of these requests be interleaved? As a part of the concurrency control issue the problem of consistency control and deadlock avoidance should be addressed -- How are the same data values of the database subject to concurrent processing by different requests to be kept consistent? How is deadlock to be avoided in an environment with multiple requests and concurrent processing?

1.1.2 Solutions for a Multi-Backend Database System Architecture

An overview of the resulting MDBS hardware organization is shown in Figure 1. The issue of backend interconnection is resolved by having the controller and backends connected by a broadcast bus. The controller will broadcast each request to all backends at the same time over this bus. Furthermore, there will be minimal broadcasting from one backend to the other backends. The issue of database store interconnection is resolved by giving each backend dedicated disk drives.

The issues of database placement and execution mode are resolved by distributing the data from each file across all the backends. Each backend will then process the data from its own disk drives. Because each file is spread across all the backends, all backends can now execute the same request in parallel. Request execution at a backend is handled by having a queue of requests at the backend. When a backend finishes executing one request it can start executing the next request. In view of the execution mode, MDBS is a multiple-instruction-and-multiple-data (MIMD) organizaton.

The data model chosen for the system is the attribute-based data model [Hsia70]. In MDBS the database consists of files of records. Each record is a collection of keywords, optionally followed by a record body. A keyword is made of an attribute-value pair such as <SALARY,$12,000> where $12,000 is the value of the attribute SALARY. A record body is a string of characters

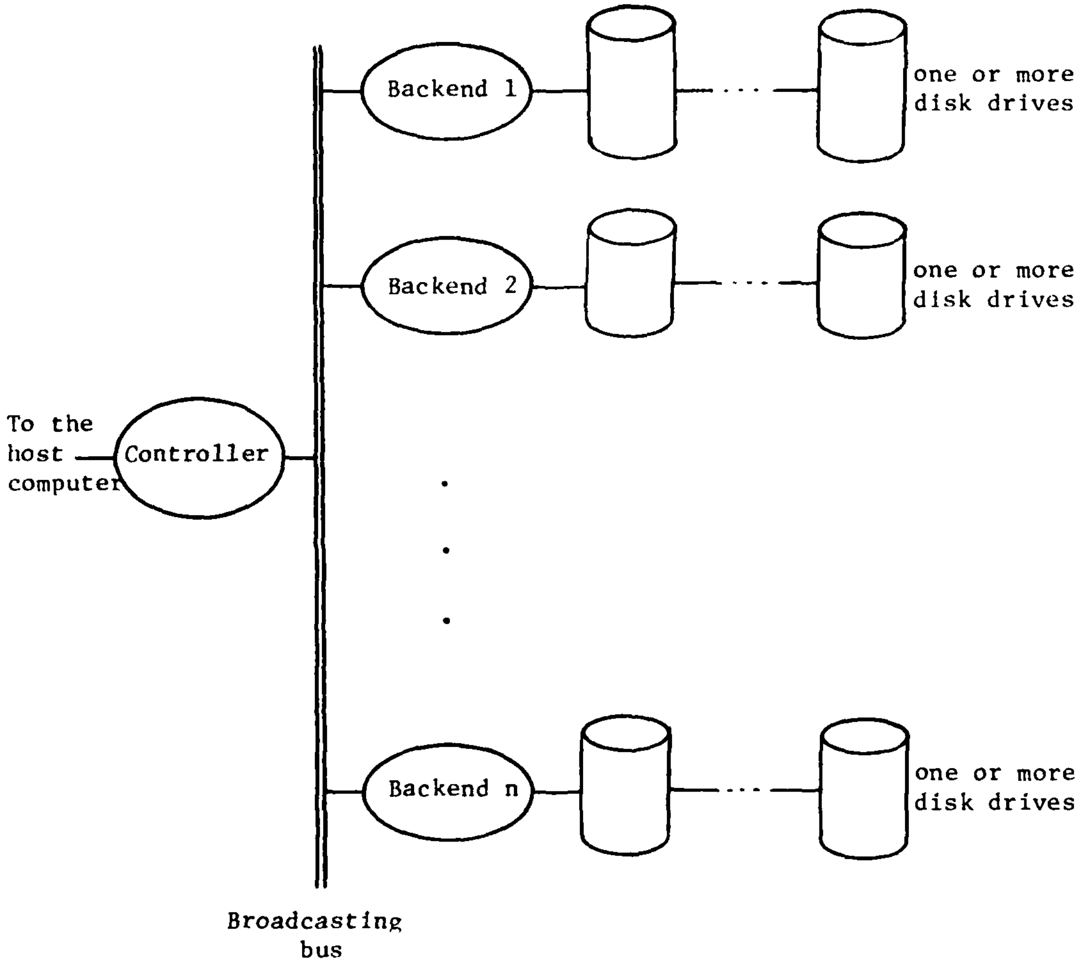


Figure 1. The MDBS Hardware Organization

not used by MDBS for search purposes. An example of a record without a record body is shown below.

(<FILE,Employee>, <EMPLOYEE_NAME,Smith>, <CITY,Columbus>,
<SALARY,$12,000>, <SERVICE,10>)

The first attribute-value pairs in all records of a file are the same. In particular, the attribute is FILE and the value is the file name. For example, the above record is from the Employee file.

For pertormance reasons, records are logically grouped into clusters based on the attribute values and attribute value ranges in the records. These values and value ranges are called descriptors. For example, one cluster might contain records for those employed in Columbus, making at least $20,001 but not more than $25,000 and with at least 11 but not more than 15 years of service. Thus records of this cluster are grouped by the following three descriptors:

(CITY=Columbus), ($20,001=<SALARY=<$25,000), (11=<SERVICE=<15).

Record retrieval in MDBS, for example, is done by clusters. Thus finding records of employees in Columbus making between $21,000 and $22,000 per year and with 12 to 13 years experience would require the retrieval of records in the cluster just described. Other requests such as to find records of employees in Columbus making between $21,000 and $28,000 and with 12 to 13 years experience might require additional retrieval of records from other clusters than the one identified above.

In order to allow efficient processing of requests, records in a cluster are spread across all the backends. Thus each backend needs to search only its portion of the cluster. Given a user request, there must be a way, of course, first to determine which clusters to search and then to determine the location of records in a given cluster. To perform this task, MDBS utilizes available descriptor information. For example, given the previous request for finding employees where

(CITY=Columbus) and ($21,000=<SALARY=<$28,000) and (12=<SERVICE=<13)

MDBS first determines that two clusters must be searched. These are the clusters identified by the two sets of descriptors:

{ (CITY=Columbus), ($20,001=<SALARY=<$25,000), (11=<SERVICE=<15) }
{ (CITY=Columbus), ($25,001=<SALARY=<$30,000), (11=<SERVICE=<15) }

After the clusters are identified, MDBS must then determine the disk addresses of the clusters at each backend. Finally MDBS will cause each backend to retrieve from its disks the records so addressed.

The execution phases of a retrieval request are summarized in Figure 2. *Descriptor search* determines the descriptors that correspond to the request. In our example, there are four descriptors corresponding to the request; namely,

(CITY=Columbus), ($20,001=<SALARY=<$25,000),
($25,001=<SALARY=<$30,000), (11=<SERVICE=<15).

In order to save space and to save processing time each descriptor is identified by a *descriptor id*. For example,

Descriptor	Descriptor Id
(CITY=Columbus)	D15
($20,001=<SALARY=<$25,000)	D125
($25,001=<SALARY=<$30,000)	D126
(11=<SERVICE=<15)	D250

Thus the output of the descriptor search phase is the Boolean expression of descriptor ids

D15 and (D125 or D126) and D250 (1)

corresponding to

$$\text{(CITY=Columbus) and} \left\{ \begin{array}{c} (\$20{,}001=<\text{SALARY}=<\$25{,}000) \\ \text{or} \\ (\$25{,}001=<\text{SALARY}=<\$30{,}000) \end{array} \right\} \text{and (11=<SERVICE=<15)}$$

which identifies two clusters.

The next phase, *cluster search* must take the Boolean expression in (1) and actually determine the corresponding clusters. As with descriptors, clusters are also identified by ids, known as *cluster ids*, for example

Descriptor Ids	Cluster Id
D15, D125, D250	C17
D15, D126, D250	C22

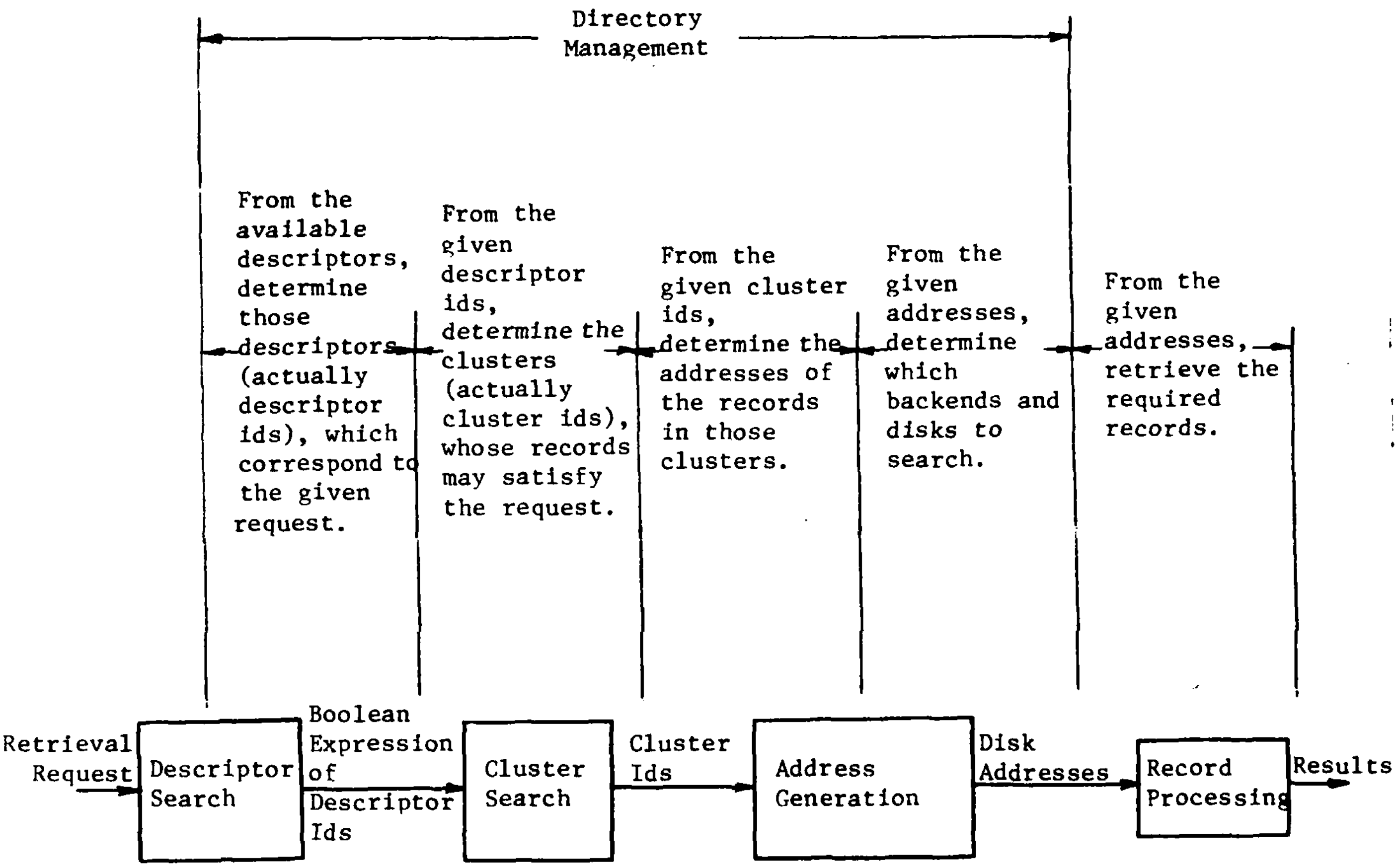

Directory Management
From the available descriptors, determine those descriptors (actually descriptor ids), which correspond to the given request.
From the given descriptor ids, determine the clusters (actually cluster ids), whose records may satisfy the request.
From the given cluster ids, determine the addresses of the records in those clusters.
From the given addresses, determine which backends and disks to search.
From the given addresses, retrieve the required records.
Retrieval Request
Descriptor Search
Boolean Expression of Descriptor Ids
Cluster Search
Cluster Ids
Address Generation
Disk Addresses
Record Processing
Results

Figure 2. Execution Phases of a Retrieval Request

The final two phases are address generation (to find the disk addresses, e.g., A3546 and A3547, corresponding to each cluster id, e.g., C17) and record selection (to retrieve the actual records so addressed).

Descriptor search, cluster search and address generation together form the major portion of directory management.

Because all directory management is based on the concept of clusters, it is also logical to design an access control capability based on clusters. Thus cluster search is augmented by a cluster access control mechanism.

The final design issue was the question of the degree of concurrency to be allowed. Executing one request at a time at a backend will frequently leave the backend's CPU idle while waiting for a disk to access records. Since the MDBS hardware organization provides multiple disk drives per backend, it is possible for a backend to support concurrent processing of requests from different users. However a mechanism to control concurrent access to data must then be provided. The mechanism used in MDBS is also centered on the concept of clusters. In particular, the concurrency control mechanism will lock clusters to prevent conflicting access to the same clustered data.

This section has described the general method used by MDBS in processing a retrieval request. This processing is summarized in Figure 3. The next section will show how this processing is divided among the controller and the backends.

1.1.3 Distribution of Request Execution Among Controller and Backends

In the previous section, we mentioned how the database was distributed across the backends. However, we did not discuss the placement of directory data and the distribution of the processing required in directory management. In order to minimize the time for directory management and to facilitate record update, the directory data is duplicated at all backends. On the other hand, the processing required for directory management is not duplicated at each backend. The descriptor search phase, instead, is divided among

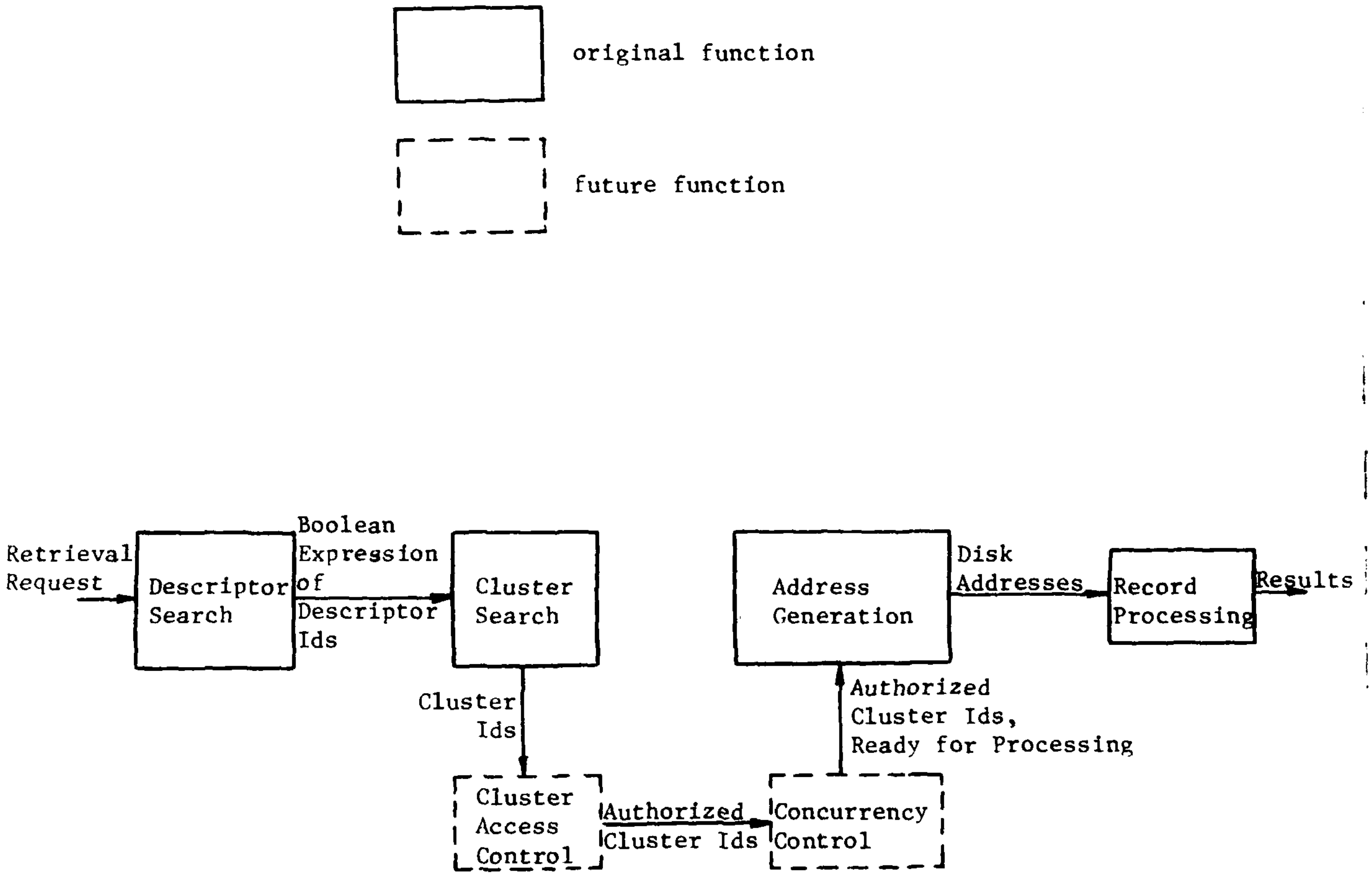


Figure 3. Execution of a Retrieval Request in the Presence of Access Control and Concurrency Control

the backends. Each backend must find only a subset of descriptor ids. It then broadcasts its results to all the other backends. In Figure 4 we summarize how directory management is performed at a backend. A retrieval request is received from the controller. Then the backend performs a descriptor search on its portion of the request and broadcasts the resulting descriptor ids to the other backends. After the descriptor ids from all other backends have been received, cluster search is used to determine the clusters. Finally, address generation determines the local disk addresses for records at that backend.

The backend can do more than just retrieve all the records in a cluster. First, it can select those records that actually satisfy the request. For example, the request to find records of employees in Columbus earning more than $20,000 but not more than $28,000 and with more than 10, but not more than 15 years experience, requires selecting records from the two clusters. Those clusters are identified by

(CITY=Columbus) and ($20,001=<SALARY=<$25,000) and (11=<SERVICE=<15)

and

(CITY=Columbus) and ($25,001=<SALARY=<$30,000) and (11=<SERVICE=<15).

All the records will be selected from the first cluster, but only records with SALARY=<$28,000 will be selected from the second cluster.

Often not all the data in a record is needed to respond to a request. In this example, only the names of the employees might be required. Thus the appropriate values must be extracted from the record. The other values may be discarded. Although not shown in this example, MDBS can perform various types of aggregate operations on a set of values instead of just returning the raw values. An example would be to find the average salary of employees who live in Columbus. Thus after selecting the appropriate records and extracting the salary values, MDBS would compute the average. The steps of record processing are summarized in Figure 5.

Referring to Figure 6, the execution of a user request can now be summarized as follows. The user submits a request to the host, which then transmits that request, in an internal form, to the controller of MDBS. The controller parses the request and then broadcasts it to the backends. The

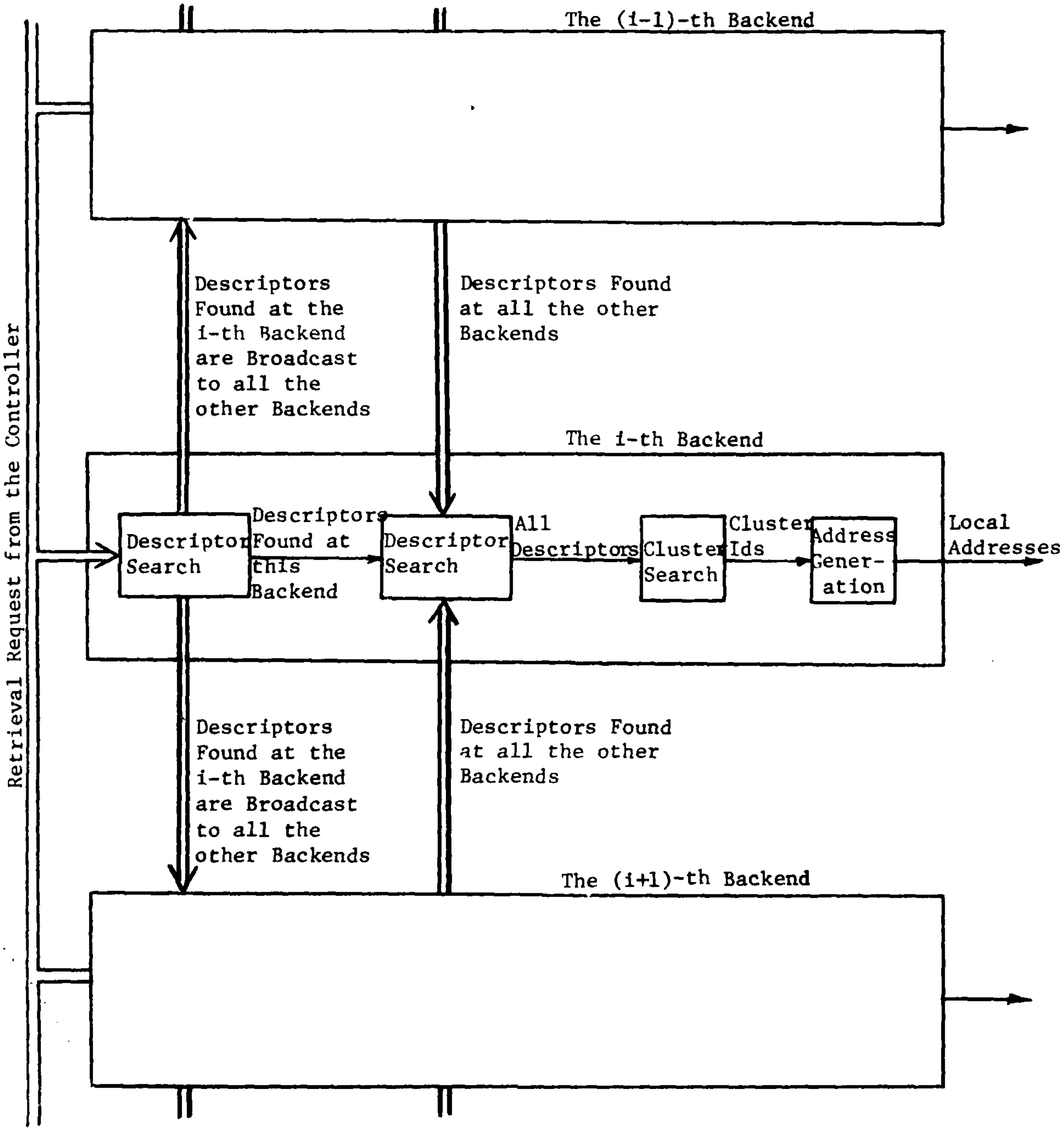

The (i-1)-th Backend
Descriptors Found at the i-th Backend are Broadcast to all the other Backends
Descriptors Found at all the other Backends
The i-th Backend
Descriptor Search
Descriptors Found at this Backend
Descriptor Search
All Descriptors
Cluster Search
Cluster Ids
Address Gener- ation
Local Addresses
Descriptors Found at the i-th Backend are Broadcast to all the other Backends
Descriptors Found at all the other Backends
The (i+1)-th Backend
Retrieval Request from the Controller

Figure 4. Overview of Directory Management as Seen From The i-th Backend

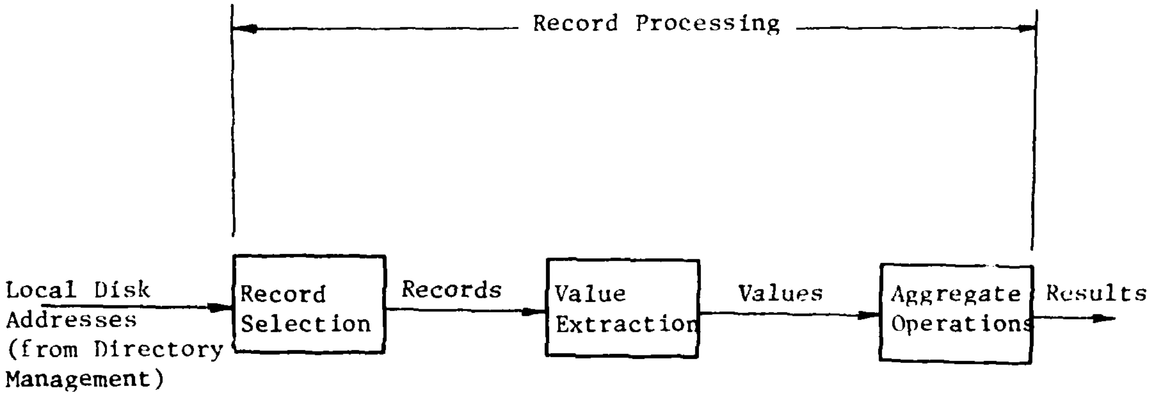
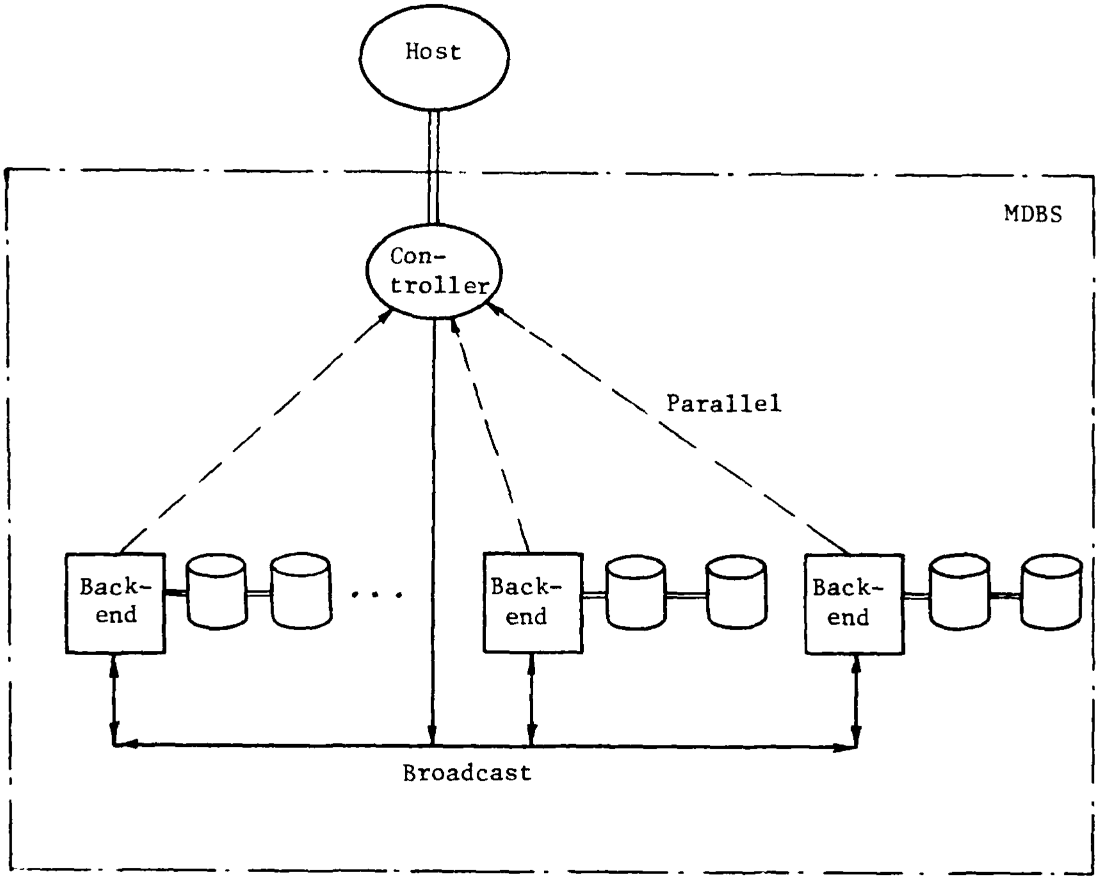


Figure 5. Record Processing Function



Broadcast Mode

- Controller-to-all-backends operation (e.g., query)
- Backend-to-all-other-backends operations (e.g., transferring descriptor ids)

Parallel Mode

- Response-of-each-backend-to-controller operations (e.g., forwarding retrieved data)

Figure 6. Modes of MDBS Operations

backends determine their portion of the descriptor ids and broadcast the results to the other backends. Each backend determines the clusters that must be searched and the corresponding local disk addresses. Then the appropriate records are selected, values extracted and results sent back to the controller. When the controller has received the results from all the backends, it performs any aggregate operation required and then forwards the final results to the host for return to the user.

1.2 Why Implement This System?

The design of MDBS is based on extensive analysis of queueing models and simulation studies of MDBS components. These results are included in [Hsia81a] and [Hsia81b]. This report is concerned with the implementation of an MDBS prototype. We, therefore, will not repeat the expected performance of MDBS as simulated and analyzed in those reports. These models and studies are, of course, only approximations. We are implementing a prototype of MDBS in order to conduct more accurate performance evaluation and more thorough design validation.

1.2.1 Validation of Simulation Results

The first reason to build a prototype system is to validate the simulation results. The main goal is to measure the extensibility of the system, i.e., how does it perform as more backends are added? In particular, is the performance gain proportional to the number of backends? If this proportionality holds for a small number of backends, how many backends can be added before no more improvement is possible? Can the response time, indeed, be improved for the same size database by increasing the number of backends, each with a smaller number of disk drives as is predicted by the simulation? The simulation models used to develop the design predict improved performance with an increase in the number of backends and the same amount of data. They also predict constant performance with an increase in data, if the number of backends is increased.

(A) System Evaluation with Program-Generated Databases

The first set of experiments will use test data that is generated by programs and specified by experimenters. The record formats will be determined by the experimenter. The actual data will then be generated from distributions specified by the experimenter. For example, one file might have 10,000 records each with 10 fields. The value in the first field of a record may be drawn from a uniform distribution on the interval [0,100]; the second field of a record may be drawn from a predefined set of values, while the third field might come from a normal distribution. The number of records and their formats can be varied in the experiments.

Requests will also be constructed in a similar way. This approach is taken first because it is easy to perform these experiments. However, we also intend to run experiments on actual databases borrowed from the Department of Defense's user community.

(B) System Evaluation With Actual Databases

The validation of the simulation will also use data taken from an actual database. Thus the second step will be to obtain one or more actual databases. Sets of "typical" requests will then be developed on the basis of the data languages of the databases. These databases and sets of requests will be used for a second set of experiments. It is hoped that such experiments will provide more insight into how a multi-backend system might actually perform. Furthermore, it will provide insights into the relative performance of the multi-backend system vs. a single-backend system and vs. a conventional system.

1.2.2 Towards a Methodology for Database Applications Classification

After experimenting with several actual databases, our goal is to develop a methodology for classifying database applications. With such a methodology it should be easier to predict the performance of a new application on an existing multi-backend system. Such a classification could also be used in the redesign of the multi-backend system, since it would allow much more accurate simulation of system performance. Right now, only two gross application classification schemes exist. One is to distinguish between "query-intensive" and "update-intensive" applications. In the first case most requests only seek information from the database, while in the second case most requests require addition and modification to the database.

A second classification scheme involves the complexity of the queries. For example, some queries are very simple, e.g., finding the address of the employee whose employee number is 123456. Other queries are much more complex, e.g., finding the names and addresses of all employees who live in Columbus, earn between $20,000 and $32,000 per year and have worked for the company for at least 10 years. There are still more complex queries which require reference to more than one file. It seems likely that some designs will provide better performance on simple queries, while other designs will provide better performance on more complex queries. These classificatons need to be made more precise. Still other classificaton schemes need to be developed.

1.2.3 Bench-Marking the System Performance

A well-known method for comparing the relative performance of computer systems is to compare the average execution time of a standard instruction mix [Ferr78]. One such mix, the Gibson mix [Gibs70], was derived from the average relative usage of IBM 7090 CPU instructions in a scientific environment. Similarly, this approach has been applied to high-level programming languages. One such mix, [Knut71], was collected for the average relative usage of Fortran statements. Once such a mix has been developed, it can be used to estimate the performance of a new computer system by first determining the execution time of each instruction type and then computing the weighted average execution time for the typical mix of instructions.

This same technique may be generalized and applied to the performance of database systems. Corresponding to a standard CPU instruction mix would be a mix of low-level database processing statements such as the requests provided by MDBS. Corresponding to the high-level programming statement mix would be a mix of high-level query language statements provided by a language such as SQL [Astr76]. The relative mix of MDBS requests or SQL statements would be determined by examining several typical database applications. This mix could be used to estimate the performance of a new database system after the execution time of each type of MDBS request or SQL statement is known.

1.3 The Implementation Strategy - What and Why?

It seems only reasonable to develop most systems in stages. For prototype systems such an approach seems even more important. Thus we plan to develop several versions of MDBS. We chose to begin with an implemention of a very simple system.

1.3.1 Version I - A Very Simple System: Single Mini Without Concurrency Control and With Simplified Directory Management

The system we are now implementing is intended to be as simple as possible. The aim is to get something running so that we can gain some experience with both the MDBS design and our new computer systems. Thus we have chosen to simplify the design as much as possible. MDBS-I will execute only a single request at a time. It will run on a single computer, i.e., a PDP11/44. There is no distinction made between the slave and master. In other words, there is no separate controller. Directory management will be simplified by storing all directory data in the main memory. There will be no concurrent execution of requests. Since the whole system will run as a single operating system process, the interface with the operating system will be minimized.

1.3.2 Version II - A Simple System: Single Mini With Concurrency Control

The second version will allow concurrent execution of requests, but will still be restricted to a single mini. We plan to use the services of our operating system to facilitate this concurrent processing. Thus we will use the capability of creating independent concurrent processes which communicate among themselves. These processes will execute in parallel so that MDBS-II will be able to execute requests in parallel. This version will allow us to gain experience with the problem of multiple processes and the problem of concurrency control.

1.3.3 Version III - The First "Real" System : Multiple Minis With Concurrency Control

After MDBS-II is working, we will transfer the system to our real environment including a controller (i.e., VAX 11/780) and several backends (PDP 11/44s). This transfer should be fairly easy, since the major changes required will be to replace communications between processes in one computer by communications between processes running on different computers. This version will allow us to see how the intercomputer communication overhead is going to affect system performance. This system, MDBS-III, will still not be sufficient for a full MDBS, since it has a very simplified directory management subsystem. However, it will allow us to begin preliminary testing of the MDBS design.

1.3.4 Version IV - The Real System With "Good" Directory Management

This version will include a fully implemented directory management subsystem utilizing the secondary memories. It will be a complete prototype system, except for the lack of access control features. This system, MDBS-IV, will be the one on which we will try to validate the simulation studies used in the development of the original design.

1.3.5 Version V - The Full System With All the Designed Features Included

The final version will incorporate access control in the backends and a friendly user-interface in the controller or host computer.

1.4 The Organization of the Rest of the Report

The rest of this report summarizes the design and implementation decisions that have been made, the software engineering approaches that have been selected and used, and the current status of the implementation.

1.4.1 Preparations for the First Effort of the Laboratory for Database Systems Research

This project marks the first implementation effort of the Laboratory for Database Systems Research. Before the implementation effort can begin, it is necessary to select the hardware to be used, both for the controller and for the backends; the implementation language; and the operating systems. The choices made and the rationale for the choices are discussed in the beginning of Chapter 2.

1.4.2 Software Engineering Approaches to the First Effort

Because the development of the prototype MDBS is our first implementation effort, we have been using this development as an exercise in implementation techniques. The actual implementation of any software system goes through several phases including specification, design, coding and testing. At present, the specifications and high-level design of MDBS have already been completed. We continue with the detailed design phase. Specific techniques for the detailed design, coding and testing phases have been adopted. These techniques are described in Chapter 2.

1.4.3 The Implementation Status

The implementation of MDBS-I is well underway. We expect the entire system to be operational in the spring of 1982. That implementation is described in detail in Chapters 3, 4 and 5. The directory management portion of the system is completed. We have also completed a utility, database load, to pertorm the loading of a database. Finally, a package to generate files of test data is also completed.

In addition to directory management, database load utility and the test file generation package, some preliminary work has been done on the approach to be taken for concurrency control. These preliminary results are discussed in Chapter 6.

2.0 THE PROJECT PLANNING AND THE IMPLEMENTATION EFFORT AND STRATEGY

Before any effort toward implementing the MDBS prototype system can begin, many decisions are required. Project planners must choose the hardware for the prototype system. In particular, they must decide upon the minicomputers for the controller and the backends. Then the systems programming language must be selected. Finally, the operating systems must be chosen. The implementors must decide on an implementation strategy. They must develop a plan not only for what is to be done, but also for how it is to be done. The "what" of this strategy is discussed in Section 1.3, which describes the five phases of the implementation strategy for MDBS. The "how" of the strategy requires the selection of software engineering techniques to be used in the implementation effort.

The primary goal of the implementation effort is to develop a prototype of MDBS to be used in database systems research. Some future directions for this research are presented in Section 1.2. Our goal also requires the software development effort to generate reliable software in as short a time period as possible without sacrificing the reliability and quality of the software. In succeeding sections of this chapter, we will document and explain the decisions made by the project planners and implementors during the preparatory stages of the MDBS implementation effort. We will show how their choices of hardware and systems software and software engineering techniques are related to the goals of the implementation effort.

2.1 The Choice of Hardware and Systems Software

Project planners have to address three fundamental questions in preparation for the implementation effort:

(1) What kind of hardware should be used in the multi-backend database system as depicted earlier in Figure 1?

(2) What systems programming language should be selected for the MDBS development effort?

(3) What operating systems should be used in order to best support the MDBS features?

We will review the alternatives which the project planners considered, and give the reasons for their choices.

2.1.1 The PDP11/34 vs. the PDP11/44 and the PDP11/70 vs. the VAX11/780

Project planners want to select hardware which satisfies the requirements of the MDBS hardware organization at the smallest price. The MDBS hardware organization is shown in Figure 1. That organization assumes that the backends are connected by a broadcast bus. It also assumes that the dedicated disk drives at the backends have the capacity to support very large databases. In addition to the MDBS design requirements, project planners must consider that the development effort for an MDBS prototype will probably require more computing power than the computing power required to run the prototype.

Since the planners anticipated an equipment grant from Digital Equipment Corporation (DEC), a proposal for DEC equipment was drafted. The proposal suggests that the most cost-effective selection of hardware would be PDP11/34s for backends and a PDP11/70 for the controller. At the time this proposal was drafted, the latest generation of the corresponding DEC minicomputers was represented by the PDP11/44 and the VAX11/780.

In a multi-backend database system, performance is improved by increasing the number of backends. MDBS is designed to be easily extensible, as explained in Section 1.1, so that no significant software development or down-time costs are incurred in expanding the system. The greatest expense incurred will be the cost of the hardware. Therefore, the cost of adding backends to a system is an important measure of cost-effectiveness. In 1979, the PDP11/34 minicomputer was the least expensive model in the PDP11 series which supports large-capacity, hard disks and can be interconnected with DEC´s Parallel Communication Bus (PCL). Using PDP11/34s as backends will minimize expansion costs for MDBS. Hardware cost is less important in selecting the controller than in selecting the backends. The PDP11/70 can furnish additional computing power required to support the development effort at a reasonable cost.

DEC's response to the original proposal was that since the ultimate goal of the implementation effort is database systems research and not product development, the latest technology available should be used. Although newer equipment may be more costly, it may also enhance the research and implementation effort. The final agreement, therefore, shows: PDP11/44s are used as the backends; the VAX11/780 is used as the controller and to support the program development effort; and the PCL is used to interconnect the VAX11/780 and PDP11/44s for the purpose of simulating the broadcast and parallel transfer capabilities. (See Figure 6 again.)

2.1.2 The Systems Programming Language

A systems programming language for the MDBS implementation must be powerful yet relatively easy to use. In other words, the language must have enough constructs to program the features for the multi-backend database system discussed earlier in Section 1.1.2. It is also important to choose the programming environment and language constructs which will make the development effort easier. The implementation team for MDBS is composed primarily of computer science students who have little practical experience, although they have a broad base in textbook knowledge. A systems programming language which makes the development effort easier will help these relatively inexperienced implementors to develop more reliable software.

Systems programming languages can be evaluated in terms of: availability, portability, and vendor support; programming environment and language features; and reliability and efficiency. Project planners examined three systems programming languages; Bliss, C, and MAINSAIL. A brief evaluation of each language and a summary of the reasons for choosing C follow this section. Some important language features and issues which are addressed in the evaluations are also explained in the following sections.*

* We wish to thank José Alegria and Tom Bodnovich for the background material which they contributed for this section.

A data abstraction is a group of related functions or operations that act upon a particular class of objects. Users of an object represented by a data abstraction are constrained to use only the operations defined in the abstraction. The classic example of a push-down stack as a data abstraction includes operations to create new stacks, to "push" data onto a stack, to "pop" data off the top of a stack, and to test for stack-full and stack-empty conditions. This technique is useful in enforcing data integrity and in controlling concurrent operations on shared data. Such a language feature will be a useful way to implement solutions for one of the design issues for a multi-backend system mentioned in Section 1.1.1, i.e. the software issue of degree of concurrency.

Another useful feature in a language is some mechanism for type-checking. Such a mechanism assures that the data types of the operands in an expression (or subexpression) are compatible with the operation which is to be performed. Type-checking contributes to the overall reliability of the software. The issue of reliability of a language involves whether or not the instructions in a language actually do what they are purported to do by the language designers and compiler writers. Clearly an unreliable language leads to unreliable software. Remember that a reliable prototype of MDBS is our goal.

(A) The Bliss Language and Its Compilers

The Bliss language [Wulf71] was originated in the Department of Computer Science at Carnegie-Mellon University. Dialects of Bliss are available from DEC for PDP11 and VAX systems, but there are significant differences between the dialects. Another limitation is that object code for the PDP11 must be generated by a cross compiler running on a larger computer system.

There is no set of programming tools for Bliss programmers, so the programming environment is poor. Bliss is an expression-level language. In its syntax, all identifiers denote addresses rather than values, so a de-reference operator ('.') must be used. For example,

```
a = .a + .b
```

is a valid Bliss instruction which, when executed, adds the values at the addresses represented by identifiers a and b and stores the result at address a. This notation makes it difficult for the uninitiated to write or read Bliss code. The language supports no primitive data types. Since operators are never type-specific, type-checking is non-existent. Nevertheless, an advantage of Bliss is that it supports the data abstraction concept.

There is some question as to the reliability of Bliss, since it contains so many low-level features. It does, however, seem to be the best of those languages surveyed when measured in terms of time/space efficiency on the DEC equipment.

(B) The C Language and Its Programming Development Environment

The C language [Kern78] was originally designed for and implemented with the UNIX operating system [Ritc74] for the DEC PDP11. UNIX is a Bell Telephone Laboratories trademark, and UNIX operating systems are licensed by Western Electric. C, however, is not tied to any particular operating system or architecture; C compilers are available on many systems. Not all versions of C are compatible, so portability can be a problem. C is supported with all versions of UNIX, and is available from the Digital Equipment Computer Users Society (DECUS), for use with PDP11 and VAX operating systems.

A rich set of program development tools usually accompanies UNIX system software. These tools provide a very good environment for C programmers. C syntax is very simple. The language supports primitive data types such as integer and character; type-checking, however, is not strongly enforced. C compilers usually do not support extra features, such as sophisticated macro processing, but many of these features are available in the programming environment support provided with UNIX. C, unlike Bliss, does not support the data abstraction concept.

C is reasonably reliable, even though many vendors do not commercially support the language. It is also reasonably efficient. A good textbook for C users is [Kern78].

(C) The MAINSAIL Language and Its Relationship to the Other Languages

MAINSAIL (MAchine INdependent SAIL) [Wilc77] evolved from the programming language SAIL, which was developed in the late 1960s at Stanford University's Artificial Intelligence Laboratory. XIDAK, Inc. owns exclusive rights to develop and market MAINSAIL. The language is distinguished by its portability. The same compiler and runtime system, both written in MAINSAIL, are the basis for every implementation; code generators and procedures which interface to the operating system must be specially written. MAINSAIL is implemented for DEC PDP11 systems.

MAINSAIL was developed and is marketed with a set of integrated tools for program development. The syntax of the language is similar to ALGOL-60. Consequently, it appears familiar to most people with formal training in computer science. The set of data types supported is more extensive than that supported either by Bliss or by C, and there is strong type-checking. On the other hand, the major disadvantage of the language is that there is no capability to invoke subroutines written in a language other than MAINSAIL or an assembly language.

Reliability is rated good. Efficiency, however, is rated lower than that of either Bliss or C. Low-level features must be coded in an assembly language, which implies that two development languages must be learned rather than one.

(D) Why Do We Choose the C Language?

The efficiency ratings of MAINSAIL and the requirement that low-level features be coded in an assembly language quickly eliminate that language from consideration. The real choice, then, lies between Bliss and C. C has a number of features which make it more desirable than Bliss.

First, C is a smaller language and has a simpler syntax. Given the inexperience of the implementation team, it is important to choose a language which can be easily learned. Next, the programming environment which can be provided under UNIX for developing C programs is a major consideration. A third factor is that the compilation process for Bliss would require the resources of a computer system at considerable distance from the Laboratory for Database Systems Research. The Laboratory initally has only two PDP11/44 systems, which are not large enough to support the Bliss compiler. The resources of a DECSYSTEM20 are available through the Department of Computer and Information Science, but it is neither convenient nor practical to set up the required communication links and procedures, since the alternative, using the C language, is acceptable. C, then, is the language we choose, since it can make the greatest contribution toward the goals of the implementation effort.

2.1.3 The Operating Systems

Important considerations in choosing the operating systems are system performance, suitability of the operating system features for the MDBS application, and suitability of the operating system features for the development effort. System performance is critical if the design goals for a multi-backend system are to be met. The first two of the three design goals, which are explained fully in Section 1.1, are:

(1) Throughput performance proportional to the number of backends.

(2) Response time inversely proportional to the number of backends.

Suitability for the MDBS application is related to the software solutions described earlier in Section 1.1.2. Operating system features must support the solutions selected for design issues such as degree of concurrency. Suitability for the development effort relates to the implementation goals to develop reliable software and to effectively manage the development effort. An operating system which is easier for relatively inexperienced programmers to use will be more suitable for development. Both UNIX and RSX11 are analyzed with these considerations in mind.

(A) The UNIX Operating System

UNIX [Ritc74] is a very "user-friendly" operating system. Interactive programs which teach the user how to use operating system facilities are a part of the UNIX package; all documentation is available on-line. A variety of aids to C programmers are available. An example is the program "Lint", which checks C programs for syntax errors, such as type violations, which are not checked by the C compiler.

The characteristics mentioned above make the UNIX environment desirable for program development. UNIX does, however, lack some system features which are required for the MDBS implementation. For example, the UNIX file system would not be satisfactory for our purposes; we would have to write a complete new input/output subsystem.

(B) The RSX11 Operating System

RSX11 is a DEC real-time operating system. Since real-time systems are engineered for execution speed, RSX11 is desirable from the performance standpoint. RSX11 also provides more flexibility; implementors can choose which operating system features to use. RSX11 also has a variety of features such as message passing which will be useful in implementing software solutions for concurrency control and backend intercommunication. RSX11 provides a less desirable programming environment than UNIX, due to the limited set of programming aids which are available through DECUS.

(C) Why Do We Choose the UNIX Operating System for the Development Effort and RSX11 for the Run-Time Effort?

The above discussions make it clear that, while UNIX is more favorable for MDBS development, RSX11 is more suitable for MDBS applications. An additional factor to be considered is that C language programs are portable from UNIX to RSX11 with only minor conversion. Furthermore, both UNIX and RSX11 are available for the PDP11/44 and the VAX11/780. Thus, project planners intend to take advantage of the best features of both systems. UNIX is to be used in the development effort, i.e., for programming the MDBS procedures. RSX11 is to be used for research purposes. The MDBS procedures when completed, will be put together and run with the RSX11 operating system as the final MDBS. It will be used to validate the results of the MDBS simulation studies - one of the research directions discussed in Section 1.2.

2.2 The "How" of the Implementation Strategy

The software development life cycle is commonly described in stages as follows:

(1) Requirements analysis

(2) Specification

(3) Design

(4) Coding

(5) Testing

(6) Operation and maintenance

It is sound software engineering practice to choose specific techniques to be used throughout the system life cycle. The software engineering objectives are to enhance the reliability of the software which is developed and to provide continuity throughout the life of the system. A further objective for the MDBS implementation effort is that the implementation should proceed as quickly and effectively as possible, since the eventual goal is to do research using the prototype system.

The MDBS implementation begins with stage two, since stage one, requirements analysis, is largely completed. The implementation strategy presented earlier in Section 1.3 details the development of five versions of MDBS. Each version after the first will be based in part upon some previous version. Furthermore, these multiple versions may not be developed in chronological order; the implementation team can be working on more than one version at the same time. Therefore, it becomes especially important to select specific software engineering techniques for the design, coding, and testing stages of the software development effort. These techniques should be selected to provide the best possible project management techniques, design and development tools, and documentation for the life cycles of all of the versions of MDBS. The techniques to be used in the MDBS implementation effort are described in succeeding sections of this report.*

2.2.1 Team Organization and Monitoring the Development Effort

Two issues in management strategy are specifically addressed in the

* We would like to thank Tamer Ozsu for the background material which he contributed for this section.

choice of software engineering techniques for the MDBS implementation effort. First, how should the group be organized? Second, what specific techniques should be adopted to monitor the development effort?

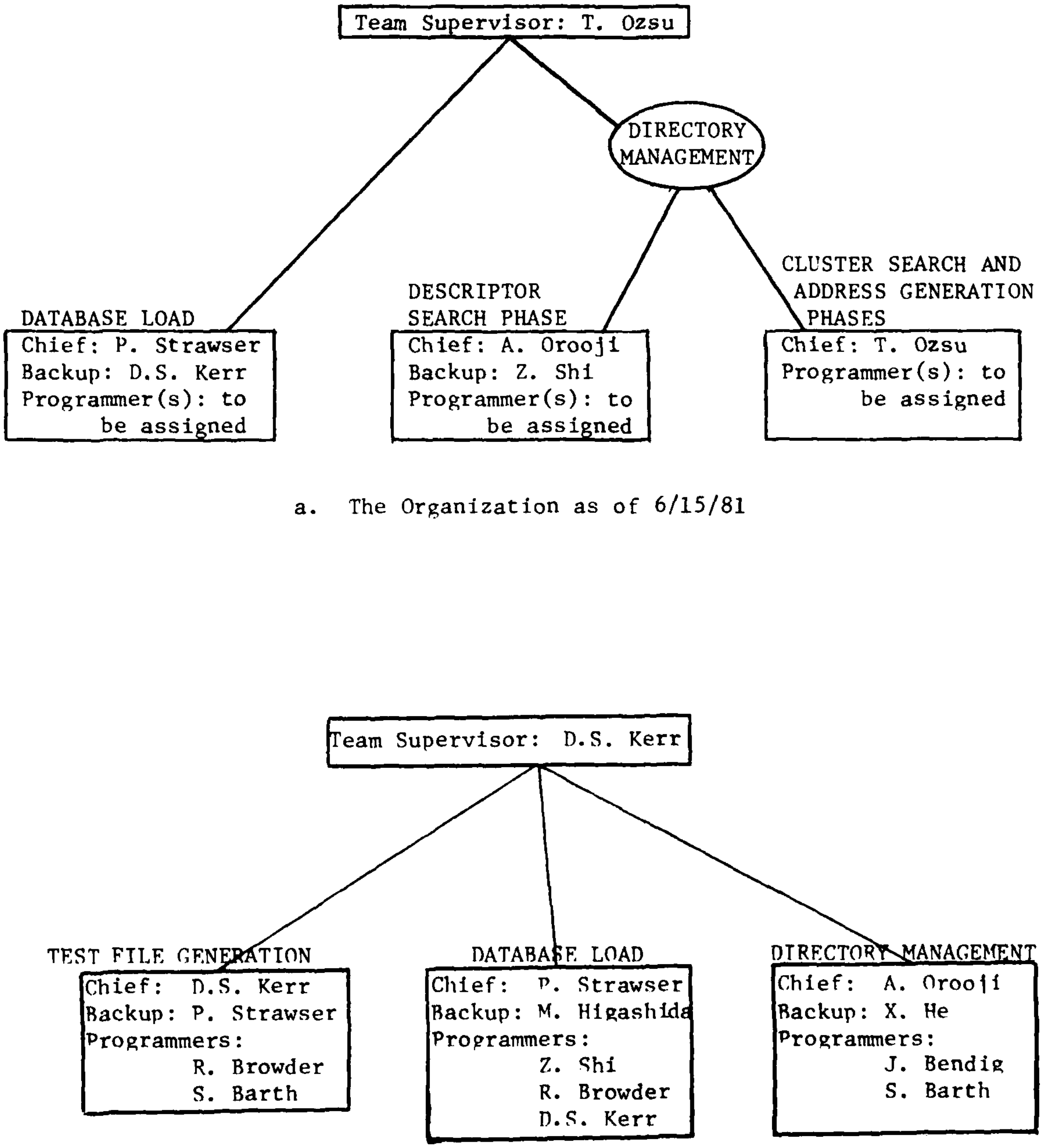
(A) A Modified Chief-Programmer-Team Organization

The classic chief-programmer team [Mill71] is headed by a project leader, the chief programmer, who has absolute decision-making authority. Other permanent members of the team include a senior-level backup programmer and a librarian. Additional programmers may be added as necessary.

The chief programmer does all the design work and writes all of the critical sections of code, for example the routines for subsystem interfaces. The backup programmer is an understudy for the chief programmer, and participates in design and coding; he takes over if the chief programmer leaves the team. The librarian maintains the group's program libraries and coordinates the documentation effort.

One advantage of such an organization is that, since the levels of communication between team members are minimized, development is likely to proceed at a faster pace than with a decentralized organization. Also, the system which is developed is likely to be more coherent and consistent since it is designed primarily by one person. By selecting this organization, we enhance the reliability and speed of development, in accordance with our software engineering objectives.

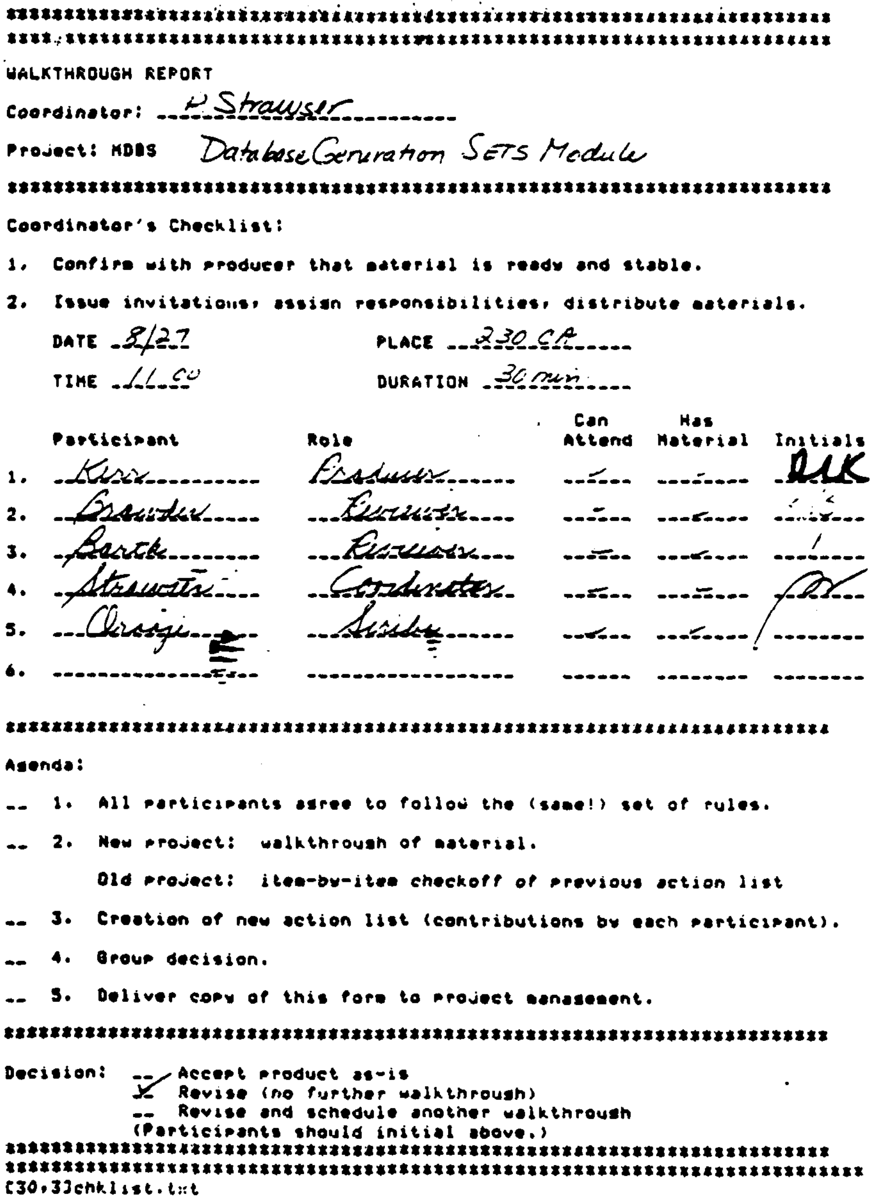
The MDBS implementation group is organized as a modified chief-programmer team. The entire effort is headed by a team supervisor. Separate teams are organized for each subproject being developed; each of these teams is composed of a chief programmer, one backup programmer and one or more programmers. A second organization chart of the group, depicted in Figure 7, shows three such teams working on directory management, test file generation, and database load.



a. The Organization as of 6/15/81

b. The Organization as of 10/1/81

Figure 7. The Organization of the MDBS Design and Implementation Teams



```
**********************************************************************
**********************************************************************
WALKTHROUGH REPORT
Coordinator: _P Strawser________
Project: MDBS   Database Generation SETS Module
**********************************************************************
Coordinator's Checklist:
1.  Confirm with producer that material is ready and stable.
2.  Issue invitations, assign responsibilities, distribute materials.
    DATE _8/27_                    PLACE _230 CA_____
    TIME _11 00_                   DURATION _30 min____
                                                   Can     Has
    Participant           Role                    Attend  Material  Initials
1.  _Kerr_________        _Producer_____           _✓___   ___✓___   DLK
2.  _Browder______        _Reviewer_____           _✓___   ___✓___   _______
3.  _Barth________        _Reviewer_____           _✓___   ___✓___   ___/___
4.  _Strawser_____        _Coordinator__           _✓___   ___✓___   _______
5.  _Orsogi_______        _Scribe_______           _✓___   ___✓___   _______
6.  ______________        ______________           ______  ________  ________
**********************************************************************
Agenda:
--  1.  All participants agree to follow the (same!) set of rules.
--  2.  New project:  walkthrough of material.
        Old project:  item-by-item checkoff of previous action list
--  3.  Creation of new action list (contributions by each participant).
--  4.  Group decision.
--  5.  Deliver copy of this form to project management.
**********************************************************************
Decision:   --  Accept product as-is
            ✓   Revise (no further walkthrough)
            --  Revise and schedule another walkthrough
            (Participants should initial above.)
**********************************************************************
**********************************************************************
[30,3]chklist.txt
```

Notes:

Refer to Figure 7a, which shows the MDBS organization chart in effect at the time this walkthrough was held. Note that three of the four chief-programmers are represented in this walkthrough committee.

This module is a part of the test file generation programming task.

Figure 8. A Sample Walkthrough Report

coding follows logically from these decisions. A top-down design strategy, implemented in a formal system specification language, and a structured coding technique are used in the MDBS implementation effort.

(A) A Top-Down Design Strategy and the Use of Data Abstraction

A top-down design strategy is a natural choice for MDBS. The design and analysis study in [Hsia81a] and [Hsia81b] clearly describes the top level of design. It also suggests the possibility of functional decomposition, i.e., the entire system can be broken into discrete functional units. This idea is supported by Section 1.1.2, which describes a multi-backend system architecture and summarizes the execution phases of a retrieval request, as depicted in Figure 2. Directory management, an example of a functional unit, includes the descriptor search, cluster search, and address generation phases of request execution.

At a lower level, one concept, data abstraction, is borrowed from the bottom-up design approach. Since MDBS is being developed as a prototype system and is to be used to research performance evaluation, we anticipate that data structures will be routinely modified in attempts to measure the effect of different data structures on system performance. The data abstraction allows us to separate the basic system functions from the data structures, minimizing the effect on the system when a data structure is modified.

(B) A Formal Systems Specification Language (SSL)

The design methodology which the MDBS implementation group uses is a systems specification language (SSL) modeled on the program description language (PDL) described in [Ling79]. The SSL adopts the same basic constructs as that PDL. The SSL is characterized by a formal "outer syntax" and an informal "inner syntax". It supports the outer-syntax constructs required for a structured design methodology - sequence, decision, and iteration. Below is an example of the if-then-else decision construct.

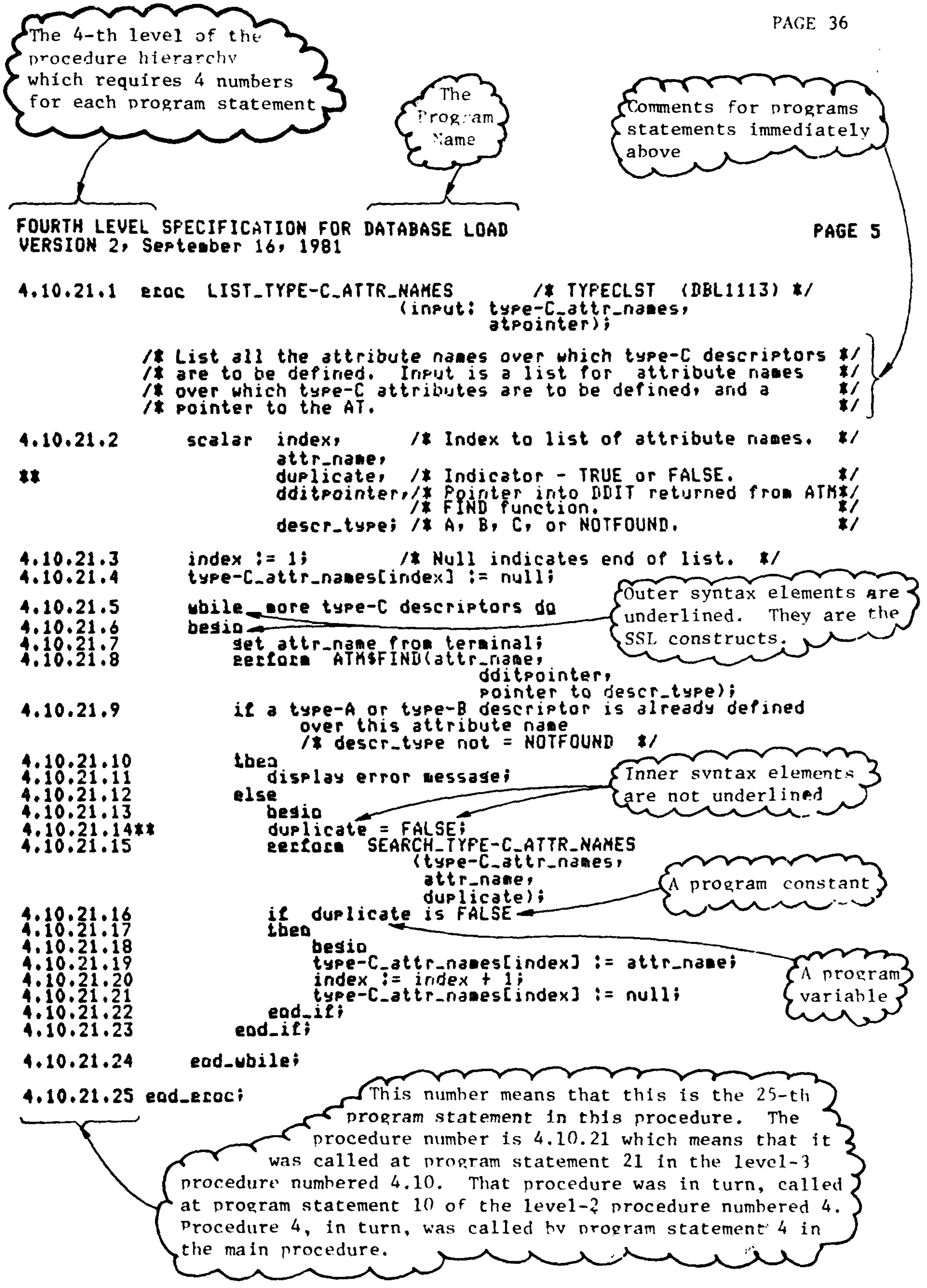
```
if expression
    then statement sequence
    else statement sequence
endif;
```

The underlined words represent the formal outer syntax. The other words represent the informal inner syntax; the only requirement for this inner syntax is that it must be understood by all team members.

In addition there are constructs for the expression of the different levels of program execution: job, module and procedure. A job is at the highest level of program execution. Test file generation described in Chapter 5 and documented in Appendix B, for example, is a job. A procedure is at the lowest level of program execution. It corresponds to the usual notion of a subroutine. Procedures are invoked to perform some work on some input data and produce some output. However, they are not allowed to retain data between invocations. Figure 9 shows a typical SSL procedure specification. More examples of SSL specifications using other constructs can be found in the appendices of this report.

Above the level of procedures, we have the level of modules. A module is intended for the implementation of a data abstraction. It consists of the procedures and data structures implementing the abstraction. An additional construct, the subsystem construct, is added to support the idea of functional decomposition. In other words, each job may perform several functions, each of which is a subsystem. Thus, subsystems are at the second highest level of program execution. Directory management described in Chapter 3 and documented in Appendix D, for example, is a subsystem, as is database load described in Chapter 4 and documented in Appendix C. The job for both directory management and database load is, of course, the MDBS.

We may also introduce one more construct, the concurrent construct, to allow the designers the capability of expressing the notion of concurrent execution, including concurrent execution at different backends. For example, directory management may be executed on all backends concurrently, while database load executes on the controller.



The 4-th level of the procedure hierarchy which requires 4 numbers for each program statement
The Program Name
Comments for programs statements immediately above

```
FOURTH LEVEL SPECIFICATION FOR DATABASE LOAD                     PAGE 5
VERSION 2, September 16, 1981

4.10.21.1   proc  LIST_TYPE-C_ATTR_NAMES        /* TYPECLST  (DBL1113) */
                                  (input: type-C_attr_names,
                                          atpointer);

         /* List all the attribute names over which type-C descriptors */
         /* are to be defined.  Input is a list for  attribute names    */
         /* over which type-C attributes are to be defined, and a       */
         /* pointer to the AT.                                          */

4.10.21.2       scalar  index,       /* Index to list of attribute names.  */
                        attr_name,
**                      duplicate,   /* Indicator - TRUE or FALSE.         */
                        dditpointer,/* Pointer into DDIT returned from ATM*/
                                    /* FIND function.                     */
                        descr_type;  /* A, B, C, or NOTFOUND.              */

4.10.21.3       index := 1;        /* Null indicates end of list.  */
4.10.21.4       type-C_attr_names[index] := null;

4.10.21.5       while more type-C descriptors do
4.10.21.6       begin
4.10.21.7           get attr_name from terminal;
4.10.21.8           perform  ATM$FIND(attr_name,
                                      dditpointer,
                                      pointer to descr_type);
4.10.21.9           if a type-A or type-B descriptor is already defined
                        over this attribute name
                        /* descr_type not = NOTFOUND  */
4.10.21.10          then
4.10.21.11              display error message;
4.10.21.12          else
4.10.21.13              begin
4.10.21.14**            duplicate = FALSE;
4.10.21.15              perform  SEARCH_TYPE-C_ATTR_NAMES
                                    (type-C_attr_names,
                                     attr_name,
                                     duplicate);
4.10.21.16              if  duplicate is FALSE
4.10.21.17              then
4.10.21.18                  begin
4.10.21.19                  type-C_attr_names[index] := attr_name;
4.10.21.20                  index := index + 1;
4.10.21.21                  type-C_attr_names[index] := null;
4.10.21.22              end_if;
4.10.21.23          end_if;

4.10.21.24      end_while;

4.10.21.25 end_proc;
```

Outer syntax elements are underlined. They are the SSL constructs.
Inner syntax elements are not underlined
A program constant
A program variable

This number means that this is the 25-th program statement in this procedure. The procedure number is 4.10.21 which means that it was called at program statement 21 in the level-3 procedure numbered 4.10. That procedure was in turn, called at program statement 10 of the level-2 procedure numbered 4. Procedure 4, in turn, was called by program statement 4 in the main procedure.

Figure 9. A SSL Specification of a Program Procedure

(C) A Practice of Structured Coding

The value of structured coding techniques to the software development effort is generally recognized. "Structured coding" refers to a methodology for problem solving as well as to the particular programming constructs used in code development.

The structured coding methodology is a top-down approach to the application of the principle of modularity, i.e., that a program procedure should have only one function. "Function" in this context means the transformation of input into output. A large problem is broken down into smaller sub-problems. This process is repeated until the solution for the smallest sub-problem is expressed as a procedure.

Structured code requires the procedure to be written with a small set of programming constructs: the statement sequence, the if-then-else and case for decisions, the do-while for iteration. It has been proved that any program can be written with only these constructs.

2.2.3 A "Black-Box" Testing Approach

In the black-box approach to testing, test data is selected without reference to the internal structure of the program. Instead, test data is generated based on the program functions described in the requirements analysis study. This approach is in contrast to the structural approach to testing, where test data is selected based on some characteristics of the internal program structure, for example, the number of paths through the program.

Intuitively, the black-box testing approach is applicable to testing database systems, since database users generally know more about the content of their databases than about the inner workings of the database system. Test data selected using the black-box approach will more closely resemble a realistic test of the system. Another advantage of the black-box approach is that, since no knowledge of internal program structures is required to develop the test data, it is easier to integrate into the testing phase the people who are not involved in the development phase.

One application of the black-box approach is functional testing [Howd80]. In this application, programs are viewed as functions which map values from the program's domain of input variables into its domain of output variables. Test data is selected based on the important properties of elements in these domains. The functional testing method is particularly suited to the MDBS implementation. The requirements analysis study in [Hsia81a] and [Hsia81b] describes the functional components of MDBS and their input and output domains. One example, explained earlier in Section 1.1.2, is the descriptor search phase of request execution. The input domain of descriptor search includes the set of retrieval requests; its output domain is the set of Boolean expressions of descriptor ids.

2.2.4 A Uniform Documentation Standard

The objectives of a uniform documentation standard are [Gilm79]:

(1) To achieve precise and unambiguous communication among staff members.

(2) To produce complete and accurate documentation.

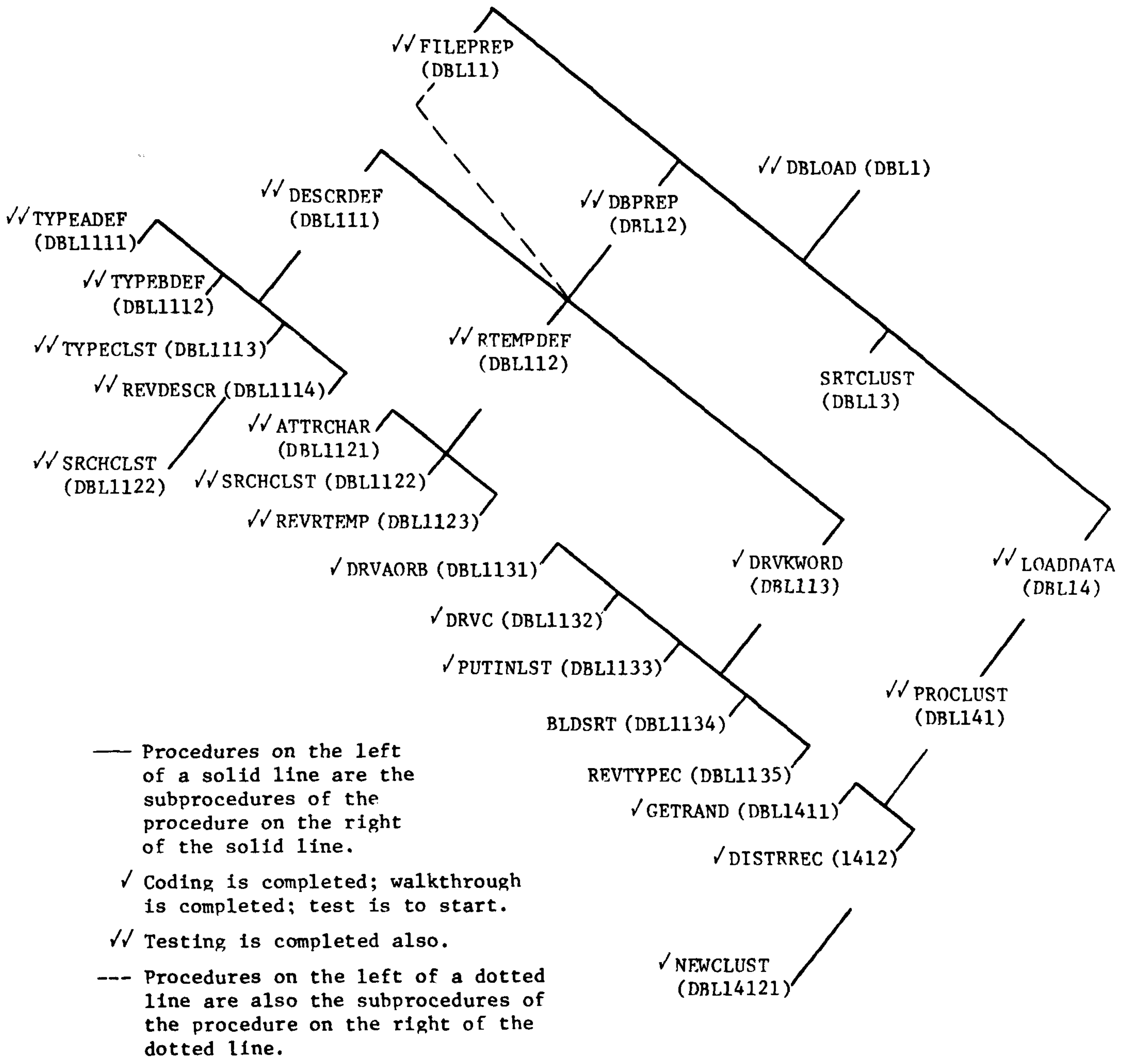
(3) To assist in project management.

(4) To reduce dependence on individuals.

We have an additional objective for the MDBS documentation standard: to integrate the documentation effort into the design and development stages of the MDBS implementation.

A documentation standard is developed in three steps. First, the terminology to be used must be selected. For MDBS, we adopt a set of standards for naming programs, program source files, and documentation text files. More specifically, each program will have a mnemonic name which describes its function as well as a coded name which identifies its place in the subsystem hierarchy. For example, the hierarchy chart in Figure 10 shows both the mnemonic and coded names for the procedures of the database load subsystem.

In the second step, the end products of the documentation effort are described. The organization and content of each document is planned in detail. For MDBS, two formal documents are proposed: a systems reference



—— Procedures on the left of a solid line are the subprocedures of the procedure on the right of the solid line.

√ Coding is completed; walkthrough is completed; test is to start.

√√ Testing is completed also.

--- Procedures on the left of a dotted line are also the subprocedures of the procedure on the right of the dotted line.

Figure 10. A Sample Procedure Hierarchy

manual (SRM), and an operating procedures manual(OPM). The SRM will be developed around the design documentation, i.e., the SSL specifications, thus minimizing the amount of new material to be written. Material for the OPM will be developed during the design of the system's user interface.

The above steps define the documentation task. The next step is to define procedures for managing the documentation effort. A documentation coordinator will assist the project manager to monitor the MDBS documentation process. Milestones in the documentation effort are identified to establish a schedule by which the coordinator can measure progress. The first of these milestones is delivery of the SSL specification to the programmer; progress of the documentation will be monitored starting at that point. A step-by-step procedure is established which charts the documentation process from the first milestone to the last milestone, which is the assembly of the finished document.

Conformity to the uniform documentation standard will assist the development group to prepare complete, accurate, and timely documentation. The MDBS implementation strategy calls for multiple versions of the MDBS prototype to be developed; some of these versions will be based on previous versions. The organization of the implementation teams is based on specific tasks; the team will be reorganized as new tasks replace completed tasks. These are two of the reasons that good documentation and a uniform documentation standard are especially important to the MDBS implementation effort.

2.3 A Retrospective

After six months experience with the MDBS implementation effort, we reexamine our decisions. Since the implementation is in its early stages, we cannot make any conclusive statements. We do, however, observe that thus far the decisions have proved to be sound. Here we will briefly review our experience with the hardware and systems software and with the software engineering techniques.

2.3.1 Evaluating the Hardware and the Systems Software

The PDP11/44s have performed as expected. The VAX11/780 is scheduled to be delivered soon. The PCL is installed and operational, although we have not yet reached a stage where the software development effort requires a broadcast capability, since MDBS-I and MDBS-II require no such capability.

To date we have not had available a working version of UNIX, so all of the development has been done under RSX11. We hope to have Berkeley UNIX on the PDP11/44s very soon. The entire implementation team is learning and using the C language as the development effort is progressing. We have encountered only those difficulties due to minimal support provided by RSX11 for programming in C. We have not yet reached a stage in system development where the underlying features of the operating system are important.

2.3.2 Evaluating the Software Engineering Experience

The project management techniques and the design and coding techniques have served us well. The SSL and the structured walkthrough have been particularly valuable. We have, however, discovered some voids in implementation of our software engineering techniques as well as some additional areas where new techniques are needed.

The largest void in implementation is that there is no project librarian to maintain code libraries and no documentation coordinator to supervise the documentation effort. An area in which the lack of any standard technique or procedure has proved to be a handicap is in the coding process, where data structures other than those encapsulated in data abstractions have been shared between subsystems. These problems can be solved, however, without invalidating any of the original decisions. It will be instructive to observe whether this remains true as the MDBS implementation progresses.

3.0 THE DESIGN AND IMPLEMENTATION OF MDBS VERSIONS

In this chapter we describe the overall designs of MDBS-I and MDBS-II. We then present the detailed designs of those parts of MDBS-I and MDBS-II that have been implemented. Occasionally, we refer to other versions of MDBS in the course of examining design alternatives. Thus, some of the design alternatives are also discussed. On the other hand, details of the implementation, i.e., data structures and program modules specified in System Specification Language(SSL), are not included in this chapter. Because they do not fit well with the designs and discussions written in the English prose, the implementation details are placed, instead, in the appendices.

In Section 3.1 we first discuss the data model used and summarize the data manipulation language adopted. As is described in Chapter 1, records are grouped into clusters by descriptors. Thus we next discuss in Section 3.2 the notion of record clustering and the use of descriptors. Finally, we summarize in Section 3.3 the entire process of request execution in MDBS-I and MDBS-II.

Section 3.4 is devoted to directory management. There, we discuss the detailed design of directory management in MDBS-I.

3.1 The Data Model and The Data Manipulation Language

In this section, we develop, in detail, the attribute-based data model used in MDBS. We then describe the data manipulation language in which users may issue requests to MDBS. The language also encompasses the useful notion of a transaction.

3.1.1 Concepts and Terminology

The smallest unit of data in MDBS is a keyword which is an attribute-value pair, where the attribute may represent the type, quality, or characteristic of the value. Information is stored in and retrieved from MDBS in terms of records. A record is made up of a collection of keywords and a record body. The record body consists of a (possibly empty) string of characters which are not used for search purposes by MDBS. For logical reasons, all the attributes in a record are required to be distinct. An example of a record without record body is shown below:

(<FILE, Employee>, <JOB, Mgr>, <DEPT, Toy>, <SALARY, 30000>).

The record consists of four keywords. The value of the attribute DEPT, for instance, is Toy. In particular, the first attribute, FILE, is known as a system attribute and the value of the system attribute is the file name of the record.

(A) Three Kinds of Keywords

MDBS recognizes several kinds of keywords: simple, security and directory. Simple keywords are intended for search and retrieval purposes. Security keywords are intended for access control. Since MDBS-I does not implement any access control feature, no reference to security keywords will be made in this report. Directory keywords are used for forming clusters. As is described in Chapter 1, records of a cluster are distributed across the backends. Within a backend, records of a cluster are stored in close proximity. We will discuss the concept of a cluster and cluster algorithms in Section 3.2.

(B) Keyword Predicates

A keyword predicate, or simply predicate, is of the form (attribute, relational operator, value). A relational operator can be one of { =, !=, >, >=, <, =< }. A keyword K is said to satisfy a predicate T if the attribute of K is identical to the attribute in T and the relation specified by the relational operator of T holds between the value of K and the value in T. For example, the keyword <SALARY,15000> satisfies the predicate (SALARY > 10000).

(C) Three Types of Descriptors

A descriptor can be one of three types:

Type-A: The descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate, such that the same attribute appears in both predicates. An example of a type-A descriptor is as follows:

((SALARY >= 2,000) and (SALARY =< 10,000)).

More simply, this is written as follows:

(2,000 =< SALARY =< 10,000).

Thus, for creating a type-A descriptor, the database creator merely specifies an attribute (i.e., SALARY) and a range of values ($2,000 and $10,000) for that attribute. We term the value to the left of the attribute the lower limit and the value to the right of the attribute the upper limit.

Type-B: The descriptor is an equality predicate. An example of a type-B descriptor is:

(POSITION = Professor).

Type-C: The descriptor consists of only an attribute name, known as the type-C attribute. Let us assume that there are n different keywords K1, K2, ..., Kn, in the records of a database with a type-C attribute. Then, this type-C descriptor is really equivalent to n type-B descriptors B1, B2, ..., Bn, where Bi is the equality predicate satisfied by Ki. In fact, this type-C descriptor will cause n different type-B descriptors to be formed. From now on, we shall refer to the type-B descriptors formed from a type-C descriptor as type-C sub-descriptors. For instance, consider that DEPT is specified as a type-C attribute for a file of employee records. Furthermore, let all employees in the file belong to either the Toy department or the Sales department. Then, two type-B descriptors will be formed as follows for this file.

(DEPT=Toy) and (DEPT=Sales)

They are the type-C sub-descriptors of DEPT.

(D) Rules for Providing Descriptors

The database creator may cause clusters to be formed for his database by giving the MDBS a list of descriptors. However, he must observe certain rules in providing the descriptors. These are specified below:

(1) Ranges specified in type-A descriptors for a given attribute must be mutually exclusive.

(2) For every type-B descriptor of the form (attribute-1 = value-1), no type-A descriptor can have the same attribute (i.e., attribute-1) and a range that contains its value (i.e., value-1).

(3) An attribute that appears in a type-C descriptor must not also appear in a type-A or a type-B descriptor defined previously.

(4) Type-A descriptors are specified first; type-B descriptors next; type-C descriptors last.

(E) The Relationship of Keywords and Descriptors

A keyword is said to be derived or derivable from a descriptor if one of the following holds:

(1) The attribute of the keyword is specified in a type-A descriptor and the value is within the range of the descriptor.

(2) The attribute and value of the keyword match those specified in a type-B descriptor.

(3) The attribute of the keyword is specified in a type-C descriptor.

(F) Query Conjunctions and Queries

A query conjunction, or simply conjunction, is a conjunction of predicates. An example of a query conjunction is:

(SALARY>25000) and (DEPT=Toy) and (NAME=Jai).

We say that a record satisfies a query conjunction if the record contains keywords that satisfy every predicate in the conjunction.

A query is any arbitrary Boolean expression of predicates. An example of a query is:

((DEPT=Toy) and (SALARY<10000)) or ((DEPT=Book) and (SALARY>50000)).

3.1.2 The Data Manipulation Language (DML)

The data manipulation language for MDBS is a non-procedural language which supports four different types of requests - retrieve, insert, delete and update. The syntax of these various requests and examples of them are presented below.

(A) Retrieve Requests

The syntax of a retrieve request is:

RETRIEVE Query Target-List [BY Attribute] [WITH Pointer].

That is, it consists of five parts. The first part is the name of the request. The second part is a query which identifies the portion of the database to be retrieved. The target-list is a list of elements. Each element is either an attribute, e.g., SALARY, or an aggregate operator to be performed on an attribute, e.g., AVG(SALARY). We will support five aggregate operators - AVG, SUM, COUNT, MAX, MIN - in MDBS. An example of a target-list of two elements is (NAME,SALARY). The values of an attribute in the target-list are retrieved from all records identified by the query. If no aggregate operator is specified on the attribute in the target-list, its values in all the records identified by the query are returned directly to the user or user program. If an aggregate operator is specified on the attribute in the target-list, some computation is to be performed on all the attribute values in the records identified by the query and a single aggregate value is returned to the user or user program. The fourth part of the request, referred to as the BY-clause, is optional as designated by the square brackets around it. The use of the By-clause is explained by means of an example. Assume that employee records are to be divided into groups on the basis of the departments for the purpose of calculating the average salary for all the employees in a department. This may be achieved by using a retrieve request with the specific target-list, (AVG(SALARY)), and the specific BY-clause, BY DEPT. Finally, the fifth part of the request, which is an optional WITH-clause, specifies whether pointers to the retrieved records must be returned to the user or user program for later use in an update request. Some examples of retrieve requests are presented below.

Example 1. Retrieve the names of all employees who work in the Toy Department.

```
RETRIEVE (FILE=Employee) and (DEPT=Toy) (NAME)
```

Example 2. Retrieve the names and salaries of all employees making more than $5000 per year.

```
RETRIEVE (FILE=Employee) and (SALARY>5000) (NAME,SALARY)
```

Example 3. Find the average salary of an employee.

```
RETRIEVE (FILE=Employee) (AVG(SALARY))
```

Example 4. List the average salary of all departments.

```
RETRIEVE (FILE=Employee) (AVG(SALARY)) BY DEPT
```

(B) Insert Requests

The syntax of an insert request is:

```
INSERT Record
```

where the Record is to be inserted into the database. An example of an insert request is:

```
INSERT (<FILE,Employee>,<SALARY,5000>,<DEPT,Toy>)
```

(C) Delete Requests

The syntax of a delete request is:

```
DELETE Query
```

where the Query specifies the particular records to be deleted from the database. An example of a DELETE request is:

```
DELETE (NAME=Hsiao) or (SALARY>50000)
```

(D) Update Requests

The syntax of an update request is:

```
UPDATE Query Modifier
```

where the Query specifies the particular records to be updated from the database and the Modifier specifies the kinds of modification that need to be done on records that satisfy the query. In an update request, if a single attribute value is tc be changed, then the attribute is termed the *attribute being modified*. The modifier in an update request specifies the new value to be taken by the attribute being modified. The new value to be taken by the attribute being modified is specified as a function f of the old value of either the same attribute or some other attribute (say, attribute-1). More specifically, the modifier may be one of the following five types:

Type-0 : <attribute=constant>
Type-I : <attribute=f(attribute)>
Type-II : <attribute=f(attribute-1)>
Type-III : <attribute=f(attribute-1) of Query>
Type-IV : <attribute=f(attribute-1) of Pointer>

Let a record whose attribute is being modified be referred to as the *record being modified*. Then, a type-0 modifier sets the new value of the attribute being modified to a constant. A type-I modifier sets the new value of the attribute being modified to be some function of its old value in the record being modified. A type-II modifier sets the new value of the attribute being modified to be some function of some other attribute value in the record being modified. A type-III modifier sets the new value of the attribute being modified to be some function of some other attribute value in another record uniquely identified by the query in the modifier. Finally, a type-IV modifier sets the new value of the attribute being modified to be some function of some other attribute value in another record identified by the pointer in the modifier.

An example of a type-0 modifier is:

<SALARY=50000>

This sets the salary in all the records being modified to 50000.

An example of a type-I modifier is:

<SALARY=1.1*SALARY>

This raises the salary in all the records being modified by 10%.

An example of a type-II modifier is:

<MONTHSAL=YEARSAL/12>

This sets the monthly salary in all the records being modified to be a twelfth of their own yearly salaries.

An example of a type-III modifier is:

<SALARY=SALARY of (FILE=Wife) and (NAME=Tara)>.

This causes the following actions to be taken by MDBS. Using the query "(FILE=Wife) and (NAME=Tara)", a record is retrieved. Then, the SALARY value of that record is obtained. This value is used for the salary in all the records being modified.

An example of a type-IV modifier is:

<SALARY=SALARY of 2000>

which modifies the salary in all the records being modified to that of the record stored in location 2000. In order to use this type of modifier, the user must have previously issued a retrieve request which had WITH POINTER option.

An example of a complete update request would be:

UPDATE (FILE=Employee) <SALARY=SALARY+5000>

which gives a $5000 raise to all employees.

3.1.3 Transactions and Consistencies

In DML, we allow the flexibility for a user to specify a set of requests for repeated execution. Such a pre-specified set of requests shall be referred to as a transaction. As in other systems, a transaction must preserve consistency. A database-creator specifies a set of assertions on the database. These assertions are constraints which must be satisfied by data in the database. For instance, since employees may not have negative salaries, an assertion on the database may require that all employees have non-negative salaries. An assertion about a database is said to be true in the database if the data in the database satisfies the constraints in the assertion. A database is in a consistent state if all the assertions made on the database by the database-creator are true in the database. Finally, a transaction is said to preserve consistency if assuming the database is in a consistent state before the transaction is executed, then immediately after the transaction has completed execution, the database must be still in a consistent state.

3.2 The Notion of Record Clusters

Record clusters are formed for the purposes of narrowing the search space and minimizing the effort needed to search for records which may satisfy a given request. In other words, by organizing a database into clusters and by maintaining information about these clusters, MDBS may readily identify those clusters whose records will satisfy the given request, thereby achieving high throughput and good response time.

Although the notion of a record cluster for the aforementioned purposes is well known, the effectiveness of clusters for throughput gain and response time improvement lies in the effectiveness of the clustering algorithm for forming clusters and the placement strategy for storing these clusters. In other words, it depends on how clusters are formed and placed. Interestingly enough, it does not depend on how clusters are used. In other words, the throughput and response time of MDBS are ´immune´ to the way the clusters are utilized. This is because every request execution by MDBS will involve the search and retrieval of clusters. Such search and retrieval can always be shown to be maximal for throughput gain and response-time improvement. Briefly, this is due to our use of the descriptors as a means to define and form clusters. As we recall, a descriptor is either a single predicate or a conjunction of predicates. We may also recall that a query in a user request is a Boolean expression of predicates. Thus, a given user request will require the retrieval of data which satisfy the predicates of the expression. Since clusters are formed by the definition of descriptors and both descriptors and queries utilize the common notion of predicates, the data retrieved for the request are actually one or more clusters. Clusters therefore become the ideal formation (or unit) of data for storage and retrieval and for performance optimization.

In the following sections, we will describe how the clusters are formed in MDBS and how they are used. We will begin with some definitions.

3.2.1 Cluster Formation

For a database, the creator of the database specifies a number of descriptors called clustering descriptors, or simply, descriptors. An attribute that appears in a descriptor is called a directory attribute. We say that a directory attribute belongs to a descriptor if the attribute appears in that descriptor.

We recall that a record consists of attribute-value pairs or keywords. For purposes of clustering, only those keywords of the record which contain directory attributes are considered. Such keywords of the record are termed directory keywords. From the rules for forming descriptors specified earlier, it is easy to see that a directory keyword is derivable from at most one descriptor. For example, consider a database with SALARY as the only directory attribute. Furthermore, let (0=<SALARY=<50000) be the only descriptor D1 on SALARY specified by the database creator. Now, consider two records, one containing the directory keyword <SALARY,25000> and the other containing the directory keyword <SALARY,75000>. Clearly, the former directory keyword is derivable from the descriptor D1 and the latter directory keyword is not derivable from D1. Hence, the latter keyword is not derivable from any descriptor in the database and we say that the directory keyword is derivable from no descriptor. Since a record may have many directory keywords, each of which will be derivable from at most one descriptor, we say that the record is derived from a set of descriptors. It is possible for a record to be derived from the empty set of descriptors. There are two such cases. In the first case, it may happen that a record does not contain any directory keyword. In this case, it is said that the record is derived from the empty set of descriptors. Thus, going back to the previous example with the single directory attribute, SALARY, and the single descriptor, (0=<SALARY=<50000), a record which does not contain any salary information (i.e., no keyword with the attribute SALARY) is said to be derived from the empty set of descriptors. The second case in which a record is derived from the empty set of descriptors is when the record does indeed contain directory keywords, but these keywords are not derivable from existing descriptors. In the previous example, a record with the directory keyword <SALARY,75000> which is not derivable from the descriptor is therefore derived from the empty set of descriptors also.

If two records are derived from the same set of descriptors, they are likely to be retrieved together in response to a user request, since these two records have keywords which are derivable from the same set of descriptors. Thus, these two records should be stored together in the same cluster. A cluster is, therefore, a group of records such that every record in the cluster is derived from the same set of descriptors. We say that a record cluster is defined by the set of descriptors from which all records in the cluster are derived.

It is easy to see that a record belongs to one and only one cluster. The reasoning is as follows. A record consists of zero or more directory keywords. If it consists of zero directory keywords, it belongs to the cluster defined by the empty set of descriptors. If the record consists of one or more directory keywords, then, the record must be derived from one and only one set of descriptors, since each directory keyword is derived from at most one descriptor. This unique set of descriptors defines the unique cluster to which the record belongs. Thus, we have used the concept of descriptor sets to partition the database into equivalence classes, namely clusters.

In order to form clusters for the records in a database, the record-to-cluster algorithm is provided to take a record and determine its cluster. For each attribute-value pair in the record, determine if the attribute is a directory attribute. If it is not, then that attribute-value pair is not used for cluster determination. If the attribute is a directory attribute, determine the descriptor, if any, from which it is derived. We refer to this descriptor, if any, as the corresponding descriptor for the given attribute-value pair. The set of corresponding descriptors for all the attribute-value pairs in a record defines the cluster to which the record belongs. By using the algorithm on every record of a database at database-creation time, we may form the record clusters of the database.

3.2.2 Cluster Determination During Request Execution

Up to this point, we have been describing the process of cluster formation. We will now explain how clusters are used during request execution.

More specifically, we will explain how to determine the cluster to which a new record belongs and how to determine the set of clusters which must be retri ·ed in order to satisfy a query for retrieval, deletion or update.

(A) Inserting Records into Clusters

During the process of cluster formation described in the previous section, MDBS uses the record-to-cluster algorithm repeatedly for determining the cluster of a record in the database. This same algorithm may now be used by MDBS to determine the cluster of a record for the record's insertion. In insertion, the <u>cluster</u> <u>definition</u> <u>table</u> (CDT) is used in order to determine the secondary memory address (addresses) of this cluster. CDT is a table maintained by MDBS. There is an entry in this table for every cluster. Each entry consists of a cluster number, set of descriptor ids defining the cluster, and addresses of the records in the cluster. A sample CDT is depicted in Figure 11.

(B) Retrieving, Deleting and Updating Records from Clusters

Let us describe how MDBS determines the set of clusters which satisfy the query in a retrieval, deletion or update request. Before we may do this, we must introduce some concepts and terminology.

Descriptor X is defined to be <u>less</u> <u>than</u> descriptor Y, if the attributes in both descriptors are the same and one of the following holds.

(1) Both descriptors are of type-A and the upper limit of descriptor X is lower than the lower limit of descriptor Y.

(2) Both descriptors are of type-B and the value in descriptor X is smaller than the value in descriptor Y.

(3) Descriptor X is of type-A and descriptor Y is of type-B and the upper limit of descriptor X is lower than the value in descriptor Y.

(4) Descriptor X is of type-B and descriptor Y is of type-A and the value in descriptor X is smaller than the lower limit of descriptor Y.

The above definition also covers the case where either X or Y is a type-C descriptor, since type-C descriptors are stored as type-B descriptors

Notes:

(1) Clusters have unique cluster numbers.

(2) No two clusters have a record in common.

(3) A cluster is defined by a set of descriptors.

(4) The keywords of the records in a cluster are derivable from the descriptors of the set defining the cluster.

(5) Two sets of descriptors defining two clusters may have descriptors in common.

Cluster Number	Corresponding Set of Descriptor Ids	Address of the Record in the Cluster
C1	D2,D3	R1,R6,R7
C2	D1,D3,D7	R4,R8
C3	D4,D6	R2,R3
	⋮	

Figure 11. A Sample of The Cluster Definition Table (CDT)

in MDBS. An exactly parallel description for the greater-than relation among descriptors may also be given.

As an example, let us assume that we are given the descriptors D1 (10000=<SALARY=<20000), D2 (0=<SALARY=<8000), D3 (SALARY=9000) and D4 (SALARY=21000). Thus, D3 is less than D1; D2 is less than D3; and D1 is less than D4.

Using the above definition of less-than and greater-than for the descriptors, we are ready to describe the algorithm for determining the corresponding set of clusters for a query in a user request. The query is assumed to be in disjunctive normal form, i.e., disjunction of conjunctions. The algorithm, known as the query-to-cluster algorithm, will proceed in three steps.

Since a query conjunction consists of predicates, we will determine, in the first step, a corresponding descriptor or a corresponding set of descriptors for each predicate. This is done as follows. If the predicate in a query conjunction is an equality predicate, then the corresponding descriptor is the one from which the keyword satisfying the predicate is derived. For example, if the predicate is (LOCATION=Napa), then the keyword satisfying the predicate is <LOCATION, Napa> and the corresponding descriptor is (LOCATION=Napa). If the predicate is either a less-than or less-than-or-equal-to predicate, it is first treated as an equality predicate and the corresponding descriptor D for that equality predicate is first determined. Then, all the descriptors less than D, along with D, form the corresponding set of descriptors for the less-than or less-than-or-equal-to predicate. If the predicate is a greater-than or greater-than-or-equal-to predicate, then it is first treated as an equality predicate and the corresponding descriptor D for that equality predicate is first determined. Then, all the descriptors greater than D, along with D, form the corresponding set of descriptors for the greater-than or greater-than-or-equal-to predicate. Thus, we have determined a corresponding set of descriptors for a predicate.

The above procedure is repeated for every predicate in the query conjunction. Thus, we will have determined a correeponding set of descriptors for every predicate in a query conjunction.

Our next step is to determine the corresponding set of clusters for a query conjunction, since a query consists of one or more query conjunctions. Let the query conjunction have p predicates. Let the set of descriptors corresponding to the i-th predicate be Si. Now, form all possible groups, where each group consists of one descriptor from Si for i ranging from 1 to p. In other words, we are forming the cross-product of Si. The reason for forming this cross-product of p sets is because a query conjunction consists of a conjunction of p predicates, each of which has a corresponding set Si of descriptors. Each element in this cross-product is termed a descriptor group which is of course a set of descriptors. Intuitively, a group defines a set of clusters whose records satisfy the query conjunction.

We now consult the cluster definition table, i.e. CDT (see Figure 11 again.) However, the definitions kept in the table may not be identical to the definitions of the groups. Without relating the descriptor groups with the descriptor sets kept in the table, we may not be able to determine the clusters involved. Thus, this second step includes the determination of whether there are descriptor sets in the table which contain a descriptor group. If there are such sets, then the clusters defined by the descriptor sets are indeed the clusters referred to by the descriptor group.

By repeating this procedure for every descriptor group in the cross-product, we are able to determine the corresponding set of clusters for a query conjunction. The entire second step which is used to dete-mine the corresponding set of clusters for a query conjunction is then repeated for every query conjunction in the query. Thus, we have determined a corresponding set of clusters for every query conjunction in the query.

The final step of the algorithm determines the corresponding set of clusters for the query from the corresponding set of clusters for each query conjunction in the query. Since the query is a disjunctio of conjunctions, the corresponding set can be simply obtained as the union of the sets of clusters for each query conjunction in the query.

3.3 The Entire Process of Request Execution

In this section, we discuss the entire sequence of actions performed by MDBS in processing the four different types of requests. We shall discuss each type of request, in turn.

3.3.1 Executing an Insert Request

The syntax of an insert request in MDBS is

INSERT Record.

The controller will first parse the request and determine that it is an insert request. Next, the controller will broadcast the request to all the backends. The backends will perform descriptor processing. At the end of the descriptor search phase, the single cluster to which the record to be inserted is known to the backend(s) whose secondary memory (memories) has (have) been accommodating the cluster. The reason that more than one backend may be involved in accommodating the cluster in consideration is that the cluster being sufficiently large has been evenly distributed by the data placement strategy over several backends′ secondary memories at the database-creation time. Consequently, MDBS must decide which backend′s secondary memory is to be used for accommodating the new record. By consulting the cluster-id-to-next-backend table (CINBT), MDBS can select the secondary memory of a specific backend for record insertion. The CINBT is created at the database-creation time by the data placement strategy. A sample CINBT is depicted in Figure 12.

3.3.2 Executing a Retrieve Request

We recall that the syntax of a retrieve request in MDBS is as follows:

RETRIEVE Query Target-list [By Attribute][WITH Pointer].

The controller will first parse the request and determine that it is a retrieve request. Next, the controller will broadcast the request to all the backends. The backends will perform descriptor processing and address generation. Upon completion, each backend has a list of secondary memory

Notes:

(1) The number of backends in a MDBS may be large, say, 6.

(2) A cluster of many records is stored in a specific round-robin way among the backends' disk drives.

(3) This table is kept up to date by MDBS as new records are inserted into the database and existing records are modified which result in changes of clusters.

Cluster Number	Backend Number of the next Backend for Inserting the Record of the Cluster
C1	B3
C2	B1
C3	B2
C4	B1
C5	B6
.	.
.	.
.	.

Figure 12. The Cluster-Id-To-Next-Backend Table (CINBT)

addresses of the tracks which contain the relevant records. These tracks are accessed by the backend. The query in the request is used to select the records from these tracks. First, the records satisfying the query are selected. If a BY-clause is specified in the retrieve request, the selected records are grouped by the values of the attribute in the BY-clause. If no BY-clause is specified in the retrieve request, all the selected records are treated as a single set. Next, for each set of selected records, the values of all attributes in the target-list are extracted from the records of the set. If no aggregate operator is specified on an attribute in the target-list, the extracted values of the set are returned to the controller. If an aggregate operator is specified on an attribute in the target-list, some computation is performed on all the attribute values in the records of the set and the results are returned to the controller. For example, to compute the average salary, each backend computes the sum of all the salaries in its set of retrieved records. It then returns this sum and a count of the number of records in the set to the controller. The controller combines the sums and counts from all the backends to give the average salary, which is returned to the user. This completes the actions performed by a backend on each set of selected records. If a WITH-clause is specified in the retrieve request, the secondary memory addresses of all selected records must also be sent to the controller by each backend.

The controller will wait for responses from all the backends. Upon receiving all the responses (i.e., attribute values, aggregate values or addresses) from all backends, the controller will forward these responses to the user that issued the retrieve request. This completes the execution of the retrieve request.

3.3.3 Executing a Delete Request

As we recall, the syntax of a delete request is

DELETE Query

The execution of this request in MDBS is similar to the execution of a retrieve request. The controller will first parse the request and determine that it is a delete request. Next, the controller will broadcast the request to all backends. The backends will perform descriptor processing and address generation. Upon completion, each backend has a list of secondary memory addresses of tracks which contain relevant records. Records of these tracks are retrieved from the secondary memory by respective backends. The query in the delete request is used to select the records which are to be deleted. The selected records are then marked for deletion. The track space occupied by the marked records is not immediately recovered. Such recovery of space will be done during database reorganization time. After the records are marked, the marked records are written back to the same tracks by each backend. If all the records in a track are marked for deletion, the address of this track is removed from all entries in which it appears in the cluster definition table (CDT). Finally, each backend will send an acknowledgement to the controller to indicate that it has finished executing the delete request. Upon receiving the acknowledgements from all the backends, the controller will inform the user or user program that the delete request has successfully been completed.

3.3.4 Executing an Update Request

The syntax of an update request in MDBS is as follows:

UPDATE Query Modifier.

We recall that the modifier in an update request specifies the new value to be taken by the attribute being modified and that it may be one of the types described below.

Type-0 : <attribute = constant>
Type-I : <attribute = f(attribute)>
Type-II : <attribute = f(attribute-1)>
Type-III : <attribute = f(attribute-1) of Query>
Type-IV : <attribute = f(attribute-1) of Pointer>

An update request containing a modifier of types 0, I or II is broadcast by the controller to all the backends. The backends will perform descriptor processing and address generation. Afterwards, each backend has a list of secondary memory addresses of the tracks containing the relevant records. These tracks are accessed by respective backends and the records satisfying the query are selected from these tracks. These are the records being modified.

Each of these records is changed according to the modifier in the update request. If the modifier is of type-0, the new value is provided in the modifier. If the modifier is of type-I, the new value is computed as a function (specified in the modifier) of the value of the same attribute. Finally, if the modifier is of type-II, i.e. of the form <attribute = f(attribute-1)>, the new value is computed as a function f of the value of the attribute-1 in that record.

Due to its change in attribute values, an updated record may remain in the same cluster to which it (more precisely, pre-updated version) belonged or it may now belong to a different cluster. In the latter case, a record is said to change cluster. Recall that a cluster is a group of records such that every record in the cluster is derived from the same set of descriptors. Thus, an updated record will belong to a different cluster only if the set of descriptors from which it is derived is different from the set of descriptors from which the pre-updated version was derived. If the attribute being modified in an updated record is not a directory attribute, the updated record continues to be derived from the same set of descriptors, since only directory attributes affect the descriptors. Hence, the updated record does not change cluster. If the attribute being modified is a directory attribute, an updated record may change cluster. If an updated record changes cluster, the pre-updated record is marked for deletion and the updated record is inserted in the appropriate cluster.

Finally, each backend will send an acknowledgement to the controller to indicate that it has finished processing the update request. When it has received acknowledgements from all backends, the controller will return a message to the user to signal successful completion of the update request. This completes the processing of an update request containing modifiers of types 0, I or II.

Now, let us describe the execution of an update request containing a type-III or type-IV modifier. Recall that these modifiers have the form <attribute = f(attribute-1) of Query> and <attribute = f(attribute-1) of Pointer>. Thus, in this case, another record must first be retrieved by MDBS on the basis of a user-provided query or pointer. After the record is retrieved, the controller will extract the attribute-1 value v from the retrieved record. It will then compute the function f (specified in the type-III or type-IV modifier) on the value v and thus obtain a new value v'. The controller will then form a type-0 modifier of the form

<attribute = v'>

where attribute is the one that appeared to the left of the equality sign in the type-III or type-IV modifier. The original type-III or type-IV modifier in the update request is now replaced with this newly created type-0 modifier. In other words, MDBS converts an update request containing a type-III or type-IV modifier to an update request containing a type-0 modifier. This update request containing a type-0 modifier may now be executed in the same manner described previously.

3.4 Directory Management

In this section, we describe the detailed design and implementation of directory management in MDBS-I.

3.4.1 The Input: Non-Insert Requests and Insert Requests

The input to directory management is either the record part of an insert request or the query part of a retrieve, delete, or update request. The three non-insert request types, namely, retrieve, delete and update, require the same directory management. However, the insert request type requires a different directory management. Thus we will describe directory management in terms of two categories: non-inserts and inserts.

We recall that the directory management in MDBS-I consists of three phases. In the first phase, MDBS determines the corresponding descriptors either for each predicate of a query in the case of a non-insert request or for each keyword of a record in the case of an insert request. In the second phase, MDBS determines either the corresponding set of clusters in the case of a non-insert request or the corresponding single cluster or a new cluster in the case of an insert request. In the third phase, MDBS determines either the addresses of clusters in the case of a non-insert request or a single address for inserting the record in the case of an insert request. (See Figure 2 again.) The following tables are used in the three phases for processing either non-insert or insert requests.

(A) Four Directory Tables: The Descriptor-to-Descriptor-Id Table (DDIT),
The Attribute Table (AT), The Cluster-Definition Table (CDT) and
The Cluster-Id-to-Next-Backend Table (CINBT)

These tables are an integrated part of the directory management. Logically, they are defined as follows:

All the descriptors defined by the database creator are stored in the descriptor-to-descriptor-id table (DDIT). There is a descriptor id associated with each descriptor. A sample DDIT is depicted in Figure 13.

There is an entry in the attribute table (AT) for every directory attribute. A pointer to the DDIT is stored with each directory attribute. The pointer points to the first descriptor whose attribute is identical to the corresponding directory attribute. A sample AT is depicted in Figure 14. Also shown in the figure is the DDIT of Figure 13. By showing these two tables together, we can easily depict the pointers of AT.

The cluster-definition table (CDT) is described in Section 3.2.2. A sample CDT is also depicted earlier in Figure 11, so we do not repeat the figure here. However, we do repeat the definition here. There is an entry in this table for every cluster. Each entry consists of the cluster number, the set of descriptor ids whose descriptors define the cluster, and addresses of the records in the cluster.

The cluster-id-to-next-backend table (CINBT) is also depicted earlier in Figure 12. A backend for record insertion is chosen on the basis of this table.

Notes:

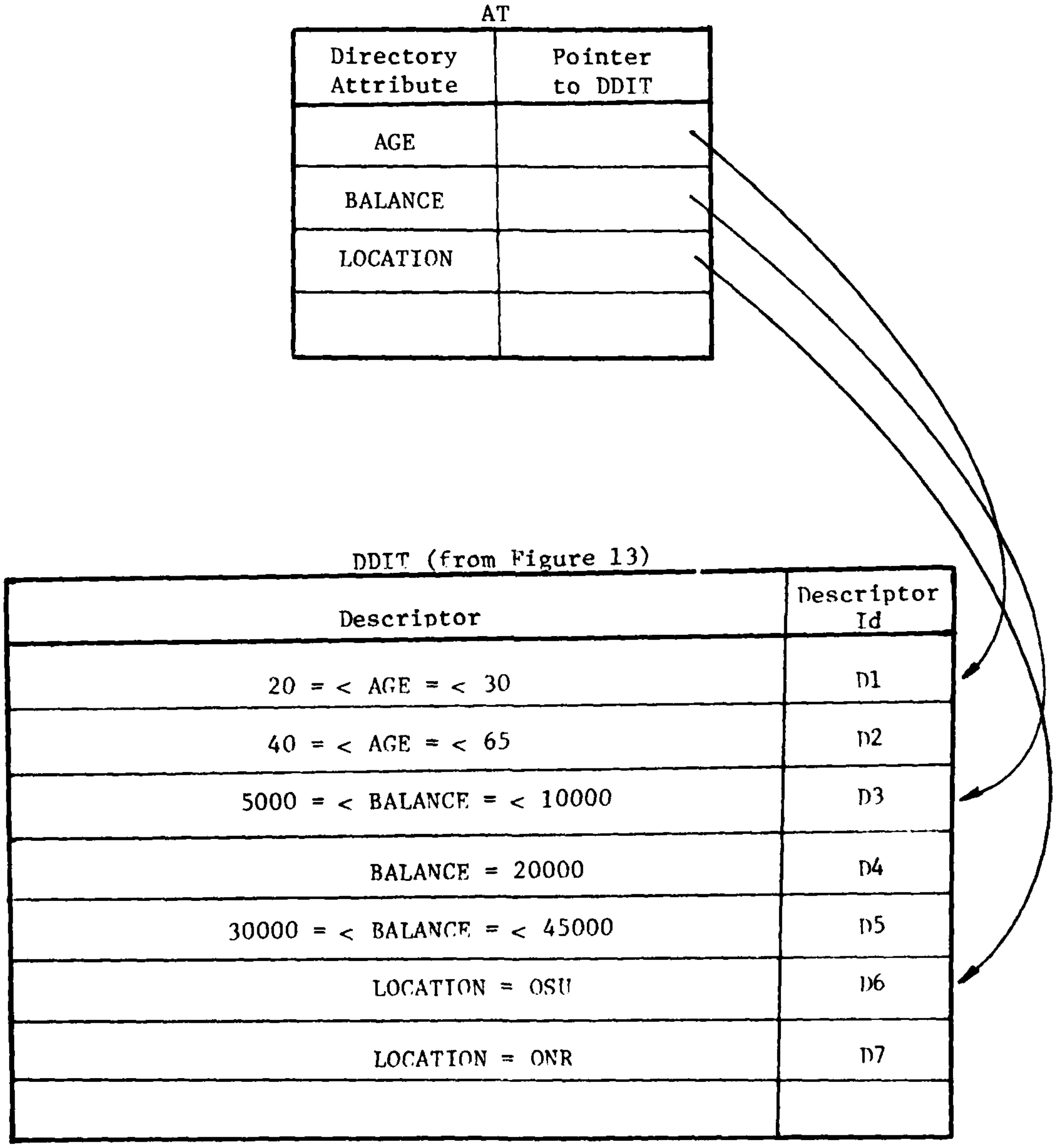
(1) Descriptors are provided by the database creator.

(2) A set of descriptors defines a cluster.

(3) Clusters are system entities which are 'transparent' to the user.

Descriptor	Descriptor Id
20 = < AGE = < 30	D1
40 = < AGE = < 65	D2
5000 = < BALANCE = < 10000	D3
BALANCE = 20000	D4
30000 = < BALANCE = < 45000	D5
LOCATION = OSU	D6
LOCATION = ONR	D7

Figure 13. The Descriptor-To-Descriptor-Id Table (DDIT)



AT

Directory Attribute	Pointer to DDIT
AGE	
BALANCE	
LOCATION	

DDIT (from Figure 13)

Descriptor	Descriptor Id
20 = < AGE = < 30	D1
40 = < AGE = < 65	D2
5000 = < BALANCE = < 10000	D3
BALANCE = 20000	D4
30000 = < BALANCE = < 45000	D5
LOCATION = OSU	D6
LOCATION = ONR	D7

Figure 14. The Attribute Table (AT) and its Relationship to DDIT

(B) Three Phases of Processing: Descriptor Search, Cluster Search and Address Generation

As described in Chapter 1, directory management has three phases. In the first phase, both AT and DDIT are searched to determine the corresponding descriptors either for each predicate of a query in the case of a non-insert request or for each keyword of a record in the case of an insert request. This is the descriptor search phase. In the second phase, the CDT is searched. For descriptors produced from the previous phase, either the corresponding single cluster in the case of an insert request or the corresponding set of clusters in the case of a non-insert request is determined. This is the cluster search phase. By searching the same CDT, the addresses of clusters can be found in the third phase. This is the address generation phase.

(C) The Choice of a Processing Strategy for the Controller and the Backends

In previous discussions, we make *no distinction* whether the three phases are carried out in a single computer (i.e., either the controller or one of the backends) or in multiple computers (a controller and several backends). In [Hsia81a], six different strategies for carrying out the descriptor search phase in the multiple backends and one strategy for carrying out the descriptor search phase in the controller are examined. There are also two strategies for carrying out the cluster search and address generation phases: one in the controller and the other in the backends.

If we are to achieve an ideal system in which the response time is inversely proportional to the number of backends, we need to distribute the directory management work among the backends. By carrying out the directory management in the backends, MDBS may be alleviated from the controller limitation problem as suggested in [Hsia81a].

In the following, we describe those three strategies that distribute the work among the backends and utilize parallel processing by the backends. All three strategies carry out the cluster search phase anc the address generation phase in all the backends. By carrying out these two phases in the backends, each backend would need to generate only those secondary memory addresses associated with that backend. On the other hand, if the addresses were to be generated by the controller, the controller would need to generate all the relevant secondary memory addresses associated with all the backends. Thus, the former case distributes address generation work among the backends; the latter case does not and concentrates all the work in the controller.

(1) The Fully-Duplicated Strategy

In this strategy, AT and DDIT are fully duplicated in all the backends. However, CDT is not duplicated. Instead, only the portion of CDT which is relevant to those clusters stored in the backend is placed in that backend. The descriptor search work is distributed among the backends. More specifically, if there are n backends in MDBS and a query contains x predicates, each backend will perform descriptor search, by using AT and DDIT, on x/n predicates and generate x/n corresponding descriptor sets which will, in turn, be communicated to all other backends. Each backend then performs, by using its portion of CDT, the cluster search phase and the address generation phase.

(2) The Descriptors-Division-Within-Attribute Strategy

In this strategy, AT is duplicated in all the backends. DDIT and CDT are not duplicated. If there are i descriptors on each directory attribute, each backend will maintain for each attribute i/n descriptors. Each backend performs descriptor search on all the predicates to generate part of corresponding descriptor sets. After each backend obtains some results, they exchange their results. Then, each backend proceeds with its own cluster search phase and address generation phase.

(3) The Fully-Replicated Strategy

In this strategy as in strategy 1, AT and DDIT are duplicated in all the backends. CDT is not duplicated. However, unlike strategy 1, each backend will work on the entire query during the descriptor search phase, instead of x/n predicates of the query. The advantage of letting each backend do the descriptor search on all predicates is that exchanges of descriptors among backends are unnecessary in this strategy because each backend has all the needed descriptors. After completing the descriptor search, each backend does its cluster search phase and address generation phase.

According to the analyses in [Hsia81a], strategy 2 has a poor average-and-worst case performance for typical number of attributes and typical number of descriptors per attribute; strategy 3 replicates the descriptor search phase; strategy 1 does not have the shortcomings of the other two strategies. Consequently, we choose to design and implement strategy 1 for directory management. In addition to utilizing strategy 1 for parallel processing of the three directory management phases for non-insert requests and the first two phases for insert requests by the backends, we choose the strategy of placing the CINBT entirely in the controller to be used only by the controller. For insert requests, the controller consults this table to select a backend for record insertion. Thus, records in a cluster can be distributed across the backends in order to achieve maximum parallel processing by the backends for subsequent requests.

3.4.2 The Use of Abstractions and Tables for Implementation

In this section, we detail the first implementation of the directory management of MDBS-I. As outlined in Chapter 1, this implementation does not provide concurrency control and access control. It maintains the directory information in the main memory only. In this implementation, cluster search and address generation are carried out together. Thus, in the sequel, we refer to descriptor search as <u>phase I</u>, and to cluster search and address generation as <u>phase II</u>. The input to phase I is either the record part of an insert request or the query part of a non-insert request, and the output is a set of descriptor ids corresponding to the descriptors derived from either the keywords of the record or the predicates of the query in the user request. Phase II makes use of these descriptor ids to come up with the corresponding cluster ids and, in turn, the set of secondary memory addresses for I/O operations.

(A) Two Data Abstractions for Descriptor Search

In compliance with the design decision of treating data structures and services, which are necessary in the phase I processing, as abstractions, both AT and DDIT tables are enclosed in data abstractions. For AT, the abstraction is the attribute-table module (ATM), and for DDIT it is the descriptor-to-descriptor-id-table module (DDITM). This approach requires access to these tables via explicit calls to procedures that operate on the tables.

(B) The Difference Between Descriptor Sets and Descriptor Groups

We now make the distinction between descriptor sets and descriptor groups by means of an example. Let us assume that MDBS has the following DDIT and CDT for the employee file:

10000 =< SALARY =< 15000	D1
20000 =< SALARY =< 30000	D2
40000 =< SALARY =< 60000	D3
20 =< AGE =< 30	D4
31 =< AGE =< 50	D5
51 =< AGE =< 70	D6
SEX = F	D7
SEX = M	D8

C1	{D2,D4,D8}	R1
C2	{D1,D5,D7}	R3,R4
C3	{D1,D4,D8}	R2,R6,R7
C4	{D3,D5,D7}	R5,R8

For this file, the descriptor set for cluster C1, for example, is

{ 20000=<SALARY=<30000 , 20=<AGE=<30, SEX=M }

Now, consider the following retrieval request.

RETRIEVE (FILE=Employee) and (SALARY>=20000) and (AGE=<50) (NAME)

In referring to DDIT, we see that the predicates of the requests have the following derivability. The predicate (SALARY>=20000) is derivable from either the descriptor (20000=<SALARY=<30000) or the descriptor (40000=<SALARY=<60000); and the predicate (AGE=<50) is derivable from either the descriptor (20=<AGE=<30) or the descriptor (31=<AGE=<50). Using their descriptor ids instead of the descriptors themselves, we learn that the query of the request is derivable from the following

(D2 or D3) and (D4 or D5)

So, for the employee file MDBS should look for clusters whose descriptor-id sets contain {D2,D4} or {D2,D5} or {D3,D4} or {D3,D5}.

To distinguish sets in CDT from those derived from the predicates, we term the aforementioned four collections of descriptor ids the descriptor-id groups and their corresponding descriptor collections the descriptor groups. For {D2,D4}, for example, the descriptor group is

{ 20000=<SALARY=<30000, 20=<AGE=<30 }

Thus, descriptor sets are associated with clusters and created either at the database creation time or when there is a new cluster, whereas descriptor groups are obtained from the query part or the record part of the request and they change from request to request. For the above retrieval request, the descriptor-id set of cluster C1 contains the descriptor-id group {D2,D4} and the descriptor-id set of cluster C4 contains the descriptor-id group {D3,D5}. Thus, the records in these clusters, i.e., {R1,R5,R8}, are retrieved, selected and the NAME values in the selected records are returned to the user.

Phase II needs descriptor-id groups to come up with cluster numbers and, in turn, addresses of the records in those clusters. In the next section, we describe how MDBS-I generates descriptor-id groups.

(C) The Generation of the Descriptor-id Groups for a Request

In order to generate the descriptor-id groups readily, we introduce the encoding scheme of location parameter. From the query part of a non-insert request, the scheme extracts the conjunctions of the query and numbers them consecutively. Each predicate is then identified by its conjunction number followed by its relative position in that conjunction. For example, in the following query part of a non-insert request

((DEPT=Shoe) and (SALARY>10000)) or ((DEPT=Toy) and (SALARY<15000))

the predicate (DEPT=Shoe) has the location parameter 11, since it is the first predicate of the first conjunction. Thus, for the above query the predicates have their location parameters represented on the left hand side:

11 DEPT=Shoe
12 SALARY>10000
21 DEPT=Toy
22 SALARY<15000

In the case of insert requests, the keywords of the record are treated as one conjunction, so the first number of the location parameter is always 1. Furthermore, the second number of the location parameter is not the relative predicate number, but the relative keyword number since the record to be inserted consists of keywords instead of predicates.

(D) A Service Abstraction for Passing Descriptor-id Groups to Cluster Search

The output of phase I, the corresponding descriptor ids, are the input to phase II. Since the format of the input to phase II depends on the cluster search strategy on CDT employed in that phase, format and strategy changes in one of the phases can affect the other phase. In order to make each phase immune to the changes made in the other phase, a service abstraction is placed between the two phases. This abstraction, known as directory interface (DIRINT), accepts the output of phase I and produces the input for phase II. All the abstractions are documented in the appendicies.

For the output of phase I, DIRINT produces a table called request descriptor-id table (RDIT), given a query part or a record part of the request. Each entry of the table is an ordered pair of location parameters and descriptor ids. Thus, an entry of RDIT indicates the id of a descriptor derived from the predicate or the keyword and is uniquely identified by the location parameter in the entry. If multiple descriptors are derived from a predicate, then there are multiple entries in RDIT, one for each such descriptor. In this case, RDIT contains the descriptor ids of all the descriptors derived from the predicate. In Figure 15, we depict a sample of RDIT.

(E) A Data Abstraction and Three Directory Tables for Cluster Search and Address Generation

In phase II, MDBS-I makes use of three tables : the descriptor table, the descriptor-to-cluster map, and the extended cluster definition table. Each entry of the descriptor table (DT) contains the id of a descriptor that has been defined for a given database, the number of clusters defined for the descriptor, and a pointer to the first cluster of those defined for the descriptor.

Location Parameter	Descriptor id
11	D4
12	D6
12	D7
12	D9
⋮	
47	D2

Multi-descriptors for the same predicate

Figure 15. A Sample Request-Descriptor-Id Table (RDIT)

The descriptor-to-cluster map (DTCM) serves the purpose of mapping descriptors to clusters. It is maintained in such a way that all the DTCM entries for a descriptor are linked together. Each DTCM entry, then, points to a cluster definition whose descriptor-id set contains the descriptor id of this descriptor.

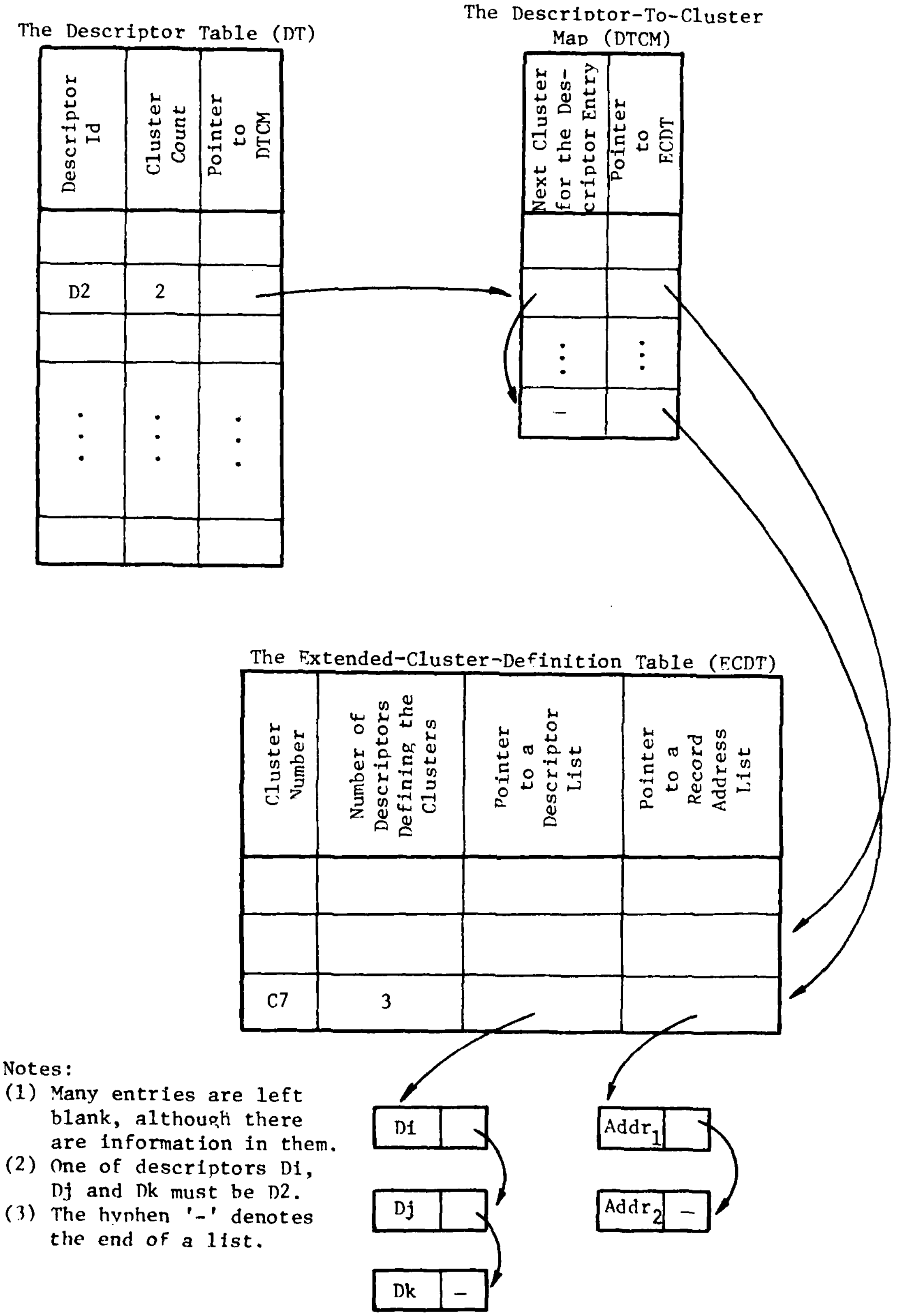
The extended cluster definition table (ECDT) contains more information about each cluster than CDT, which was discussed in Section 3.2.2 and depicted in Figure 11. Each entry consists of the cluster number of a cluster, number of descriptors defining the cluster, a pointer to the list of descriptor ids whose descriptors define the cluster, and a pointer to the list of addresses of records belonging to this cluster.

All of these tables are enclosed within a data abstraction called cluster-definition-table module (CDTM). A sample of the tables is depicted in Figure 16.

(F) A Typical Sequence of Directory Management Actions for an Insert Request

When there is a request for inserting a record, the following directory management takes place in MDBS-I. An equality predicate is constructed for each keyword of the record. For example, the keyword <NAME=Kerr> becomes the predicate (NAME=Kerr). Then, for each predicate, the descriptor id of the descriptor derived from the predicate is found by using AT and DDIT. This process is repeated for every keyword of the record. All the descriptor ids are then put into RDIT via the service abstraction DIRINT.

The descriptor-id group corresponding to the record being inserted is obtained from RDIT via DIRINT. We note that there is only one descriptor-id group because each of the equality predicates constructed from the keywords is derived from at most one descriptor. Among the descriptor ids in the descriptor-id group, the id of the descriptor that participates in defining the smallest number of clusters is chosen by using DT. Let us call this descriptor id Dm. By using DT, DTCM, and ECDT, all the clusters whose descriptor-id


The Descriptor Table (DT)
Descriptor Id
Cluster Count
Pointer to DTCM
D2
2
The Descriptor-To-Cluster Map (DTCM)
Next Cluster for the Descriptor Entry
Pointer to ECDT
–
The Extended-Cluster-Definition Table (ECDT)
Cluster Number
Number of Descriptors Defining the Clusters
Pointer to a Descriptor List
Pointer to a Record Address List
C7
3
Di
Dj
Dk
–
Addr1
Addr2
–

Notes:
(1) Many entries are left blank, although there are information in them.
(2) One of descriptors Di, Dj and Dk must be D2.
(3) The hyphen '-' denotes the end of a list.

Figure 16. An Example of DT, DTCM and ECDT

sets contain Dm are examined. If there is a cluster whose descriptor-id set matches the descriptor-id group, then the record being inserted belongs to the cluster identified. We note again that for an insert request, the descriptor-id set must match the descriptor-id group so that the record may be inserted in the cluster whose records are derived from the same set of descriptors.

(G) A Typical Directory Management Sequence of Actions for a Non-insert Request

When there is a non-insert request, the following directory management takes place in MDBS-I. For each predicate in the query part of the request, all the descriptor ids of the descriptors derived from the predicate are found by using AT and DDIT. All the descriptor ids are put into RDIT via the service abstraction DIRINT.

Each of the descriptor-id groups corresponding to the query is obtained from RDIT via DIRINT. We note that there may be more than one descriptor-id group because each predicate of the query may be derived from more than one descriptor. See the example in part (B) of this section. Among the descriptor ids in the descriptor-id group, the id whose descriptor participates in defining the smallest number of clusters is chosen by using DT. This descriptor id is designated with Dm. By using DT, DTCM, and ECDT, all the clusters whose descriptor-id sets contain Dm are examined. The clusters whose descriptor-id sets contain the descriptor-id group are therefore found. This process is repeated for each descriptor-id group. We note that for a non-insert request, the descriptor-id set does not have to be identical to the descriptor-id group as long as the set contains the group. Then, the addresses of the records in the clusters just found are obtained.

4.0 LOADING THE DATABASE

In MDBS, as in other database systems, a database creator may want to load a database with data that exists elsewhere. Such data may reside as files on magnetic tapes, for example. The database creator can use a software tool, provided by MDBS and called database load, to specify the source data files and to create a database. In this section, we describe the design of this tool. The implementation details for the version used in MDBS-I are placed in Appendix C.

4.1 Three Directory Tables for Loading

A user of the database-load subsystem may want to consolidate several related files into one database. In this case, there will be one attribute table (AT), one descripter-to-descripter-id table (DDIT) and one extended-cluster-definition table (ECDT) for the database. Alternatively, the user may want each file to become a separate database. In this case there will be a separate AT, DDIT and ECDT for each database.

4.2 Four Phases of Database Loading

The database load subsystem, as seen by the user and shown in Figure 17, executes in four logical phases. First, the user specifies various characteristics of the existing source files and of the database to be created and loaded. Then the data is read from user supplied source-files and prepared for loading. Next, the data is grouped into clusters. After clustering, the data is distributed to the backends. The programs in the database-load subsystem run mainly in the controller. However,the database-load subsystem does include the distribution of records and directory tables to the backends.

4.2.1 The Database Definition Phase

Before the source files are read, two tasks are accomplished in this phase. The first task is descriptor definition. In this task the user spec-

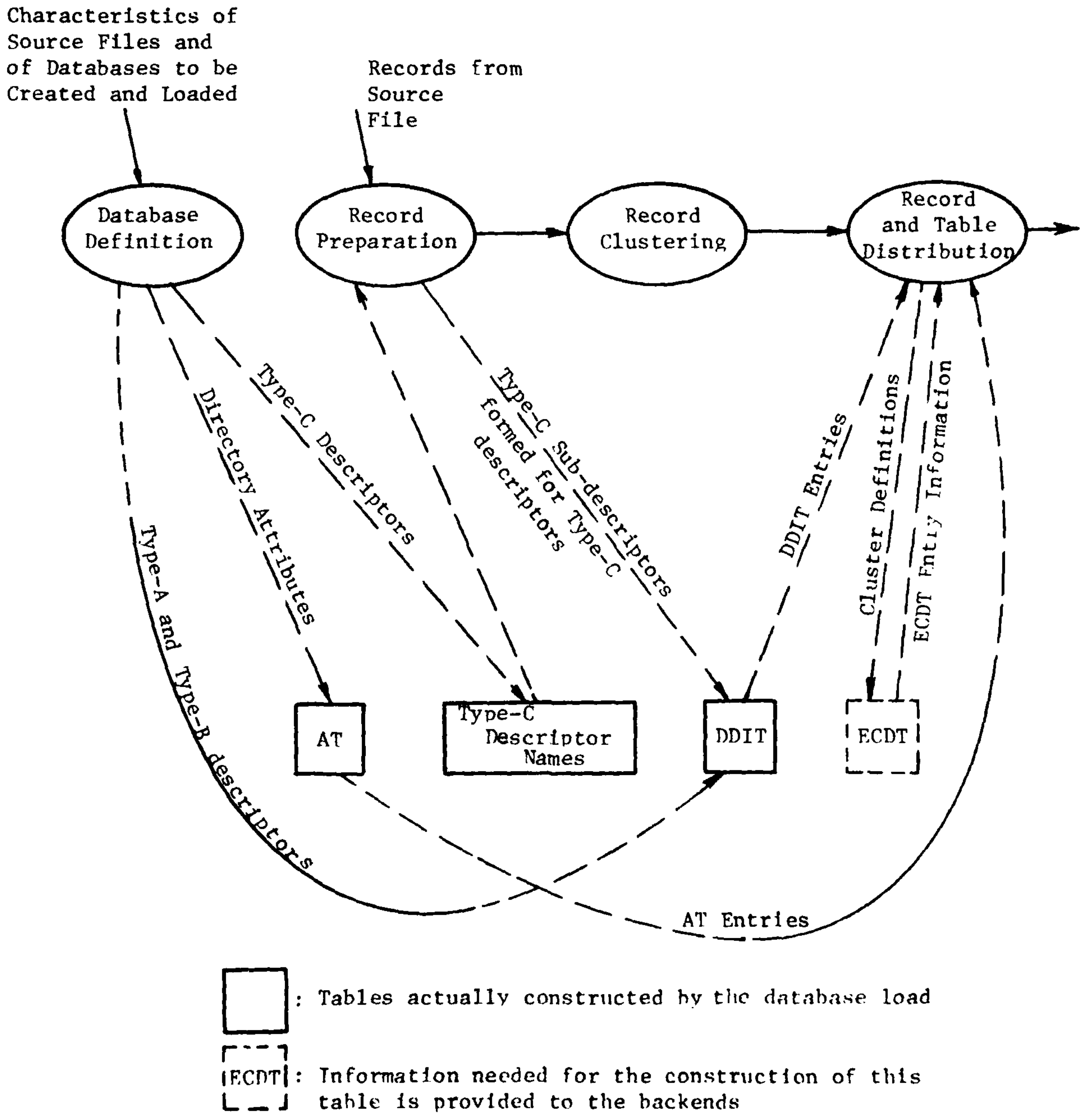

Characteristics of Source Files and of Databases to be Created and Loaded
Records from Source File
Database Definition
Record Preparation
Record Clustering
Record and Table Distribution
Type-A and Type-B descriptors
Directory Attributes
Type-C Descriptors
Type-C Sub-descriptors formed for Type-C descriptors
DDIT Entries
Cluster Definitions
ECDT Entry Information
AT
Type-C Descriptor Names
DDIT
ECDT
AT Entries
: Tables actually constructed by the database load
ECDT
: Information needed for the construction of this table is provided to the backends

Figure 17. Four Phases of Database Loading

ifies all directory attributes for the database. Then the user specifies the upper and lower bounds for each type-A descriptor and the value for each type-B descriptor. As these values are given, they are checked against previously defined descriptors to make sure there is no overlap of the ranges and values specified. In other words, the rules governing the proper use of descriptors given in Section 3.1.1 are enforced by the database load subsystem.

Only the attribute names of type-C descriptors are specified at this time. The type-C sub-descriptors of the type-C descriptors will be formed later when the actual source records are processed. As described above, clusters may be formed for one file at a time or for all files at the same time. When clusters are formed for all the files, the descriptor definition procedure will be invoked only once per database; when clusters are formed for separate files, the descriptor definition procedure will be invoked once per file.

During execution of this task the attribute table (AT) will be built using the data abstraction of the attribute table module (ATM). In addition, descriptor-to-descriptor-id table (DDIT) entries for all type-A and type-B descriptors will be established using the data abstraction of the descriptor-to-descriptor-id module (DDITM). The ATM and DDITM are described in Section 3.4.2. The type-C sub-descriptors formed from the type-C descriptor entries will be added later as the source-data is examined in the next phase.

The second task is <u>definition of attribute characteristics</u> for each file. A file is defined to include records of one format only. A <u>record template</u> will be built for each file. It will include an entry for each attribute. Each entry will include the attribute name, data-type (e.g., integer), length, etc. For each source file, the user must supply the names of all the attributes in the records. Then for each attribute, the user must define the data-type. If the data-type is character string, the user must specify whether the strings are of fixed or variable length. The user must also specify the minimum and maximum values of each integer type as well as the minimum and maximum lengths of each character string type. All of the values specified are stored in the record template by the record template module.

4.2.2 The Record Preparation Phase

This phase includes the conversion of source records into the format required for internal storage in MDBS. As each record is examined the set of descriptors from which the keywords of the record can be derived will be determined using the ATM and DDITM abstractions. In conjunction with this task, the type-C sub-descriptor entries formed from each type-C descriptor will be added to DDIT. Additionally, the formatted record will be appended to the descriptor-id set corresponding to the descriptors derived from the record. Both the descriptor-id set and their appended records are the input to the next phase. At the end of this phase the attribute table (AT) and the descripter-to-descripter-id table (DDIT) are complete.

4.2.3 The Record Clustering Phase

This phase separates the records into clusters. As is described in Chapters 1 and 3, all the records in a cluster are derived from the same set of descriptors. Thus, separating the records into clusters is accomplished by sorting the records according to the descriptor ids appended to each record in the previous phase. A sort package is used for this phase.

4.2.4 The Record and Table Distribution Phase

The last phase is distribution of data to the backends. The records are distributed to the backends one cluster at a time. For each cluster, the descriptors defining the cluster are broadcast to the backends so that the cluster can be defined in the extended-cluster-definition table (ECDT) using procedures in the cluster-definition-table module (CDTM) that was described in Section 3.4.2. Typically, a cluster will contain many records. Within a cluster, the records are spread across the backends. Sufficient records to fill one disk track are sent to one backend. Then sufficient records to fill a second track are sent to second backend. This procedure continues until all the records have been distributed. It should be noted, of course, that the last group of records may not fill a track. The information about the last backend and amount of track space available is kept in the cluster-id-to-next-backend tabl. (CINBT) so that the next records to be inserted into that cluster can be stored in that partially filled track.

In order to distribute the data evenly across all the backends, the first backend to receive records is chosen randomly. Then the choice of backends goes in sequence. This distribution strategy was called the track-splitting-with-random-placement strategy in [Hsia81a].

This phase also distributes the system tables to the backends. The attribute table (AT) and the descriptor-to-descriptor-id table (DDIT) are complete after the record preparation phase. Portions of the extended-cluster-definition table (ECDT) will be built at each backend. The portion of ECDT at a backend will contain only the addresses of the records stored in that backend.

4.3 The Implementation Status

The complete design of the database load subsystem for MDBS-I is included in Appendix C. Coding has been completed for almost all the procedures. Testing is completed for about half of the procedures. As described in Appendix A, Appendix C shows exactly which procedures have been completed and which procedures have been tested. This information is also included in Section 2.2.4 as Figure 10.

5.0 THE TEST FILE GENERATION

Program-generated test data will be used for two purposes. First, we will be testing each version of MDBS to see that it works correctly. Second, as described in Chapter 1, the initial performance evaluation experiments will use program-generated data.

5.1 Three Types of Test Data

The test data to be generated will be organized into files. Thus we designate the program the File Generaton Package. The characteristics of a file are specified by the user of the file generation package. Each file has a file-name and a certain number-of-records. Each record in a database is composed of a set of attribute-value pairs. For initial testing purposes, we have decided to require that all records in a file have the same attributes. Thus each record has a fixed number-of-attributes. The values of an attribute in different records are restricted to a particular data-type. The possible data-types are integer, string (i.e., character-strings) and float (i.e., floating-point numbers).

In addition to specifying the format of data to be generated, we must also specify how particular values of each record are to be generated. The first means of generating a value is to use a routine that generates a random-integer, random-string or random-float value. These routines make use of random-number generators to arrive at a value from some particular distributions of potential values. Thus the values of the first attribute might be a randomly chosen integer between 100 and 500. The value of the second attribute might be a random character string. The value of the third attribute might be a random floating-point number.

5.2 Random Test Data vs. Realistic Test Data

The data generated as just described is fine for program testing and initial performance evaluation. However, since each value is generated randomly by a program, the test data is not selectd by the user. In order to generate realistic data for the user, a second form of value generation is also supplied. In this form, a user may specify the data sets, say, a set of names. Then the user can direct the file generation package to select values for an attribute from one of these predefined sets. Once a set is defined, its values are saved for later use.

5.3 Steps in Test File Generation

The file generation package works in three steps. First, the user defines the form of the file to be generated, i.e., the number-of-records, number-of-attributes and the characteristics of each attribute. Then initial processing of test data sets follows. If the user wants to use sets that already exist, then the data of those sets are loaded into the main memory. If there are new sets that the user wants to specify then the program prompts for the values of the data of the sets, which are then loaded into the main memory and also stored in the secondary memory for later use. After all the sets are loaded into the main memory, the final step is the actual generation of the records.

5.4 The Relationship of the Package to Testing Strategies and Performance Evaluation Experiments

The first use for the file generation package is for the black-box testing of MDBS as described in Section 2.2.3. In particular, the system testers will be able to generate easily any form of test databases that they require. They will then only have to generate sample requests in order to run tests to see if MDBS is working correctly.

The second use for the file generation package is for the type of performance evrluation experiments using program-generated data as described in Section 1.2.1(A). For these tests, the experimenters can vary the form of the database by varying the distribution of different types of data. They can see how MDBS performs on different types of queries and using different numbers of backends.

5.5 Current Status of the Package

The file generation package is now working in its initial form and is ready to be used for black-box testing. The package handles integer and string data types. The subsystem which handles data sets is finished. The routines to generate data values from a uniform distribution are complete. The data type, float, must still be added. Routines to generate data values from distributions other than uniform are not needed for black-box testing. They will be added if it is determined that they are needed for performance evaluation experiments.

The design of the complete file generation package is included in Appendix B. The first version of the operating procedures manual (OPM) is also completed.

6.0 PLANS FOR THE NEXT MDBS VERSIONS

As we recall from Chapter 1, MDBS-I will not provide concurrency control. We also note that MDBS-I does not provide a secondary-memory-based directory management. Instead, MDBS-I utilizes the main memory for directory management. We plan, therefore, to implement a concurrency control mechanism in MDBS-II and an efficient directory management utilizing the secondary memory for MDBS-IV.

The basic design of the concurrency control mechanism is included in [Hsia81b]. We will not elaborate on the basic design here except to note that the detailed design eliminates any need for communication among the backends other than the required exchange of descriptors during the descriptor search phase of directory management. In this section, we will discuss one of most important system issues which must be resolved before we can implement the concurrency control mechanism, i.e., how will MDBS interface with the operating systems at the controller and at the backends? On the other hand, we will not discuss various approaches toward an efficient directory management based on the secondary memory since our preliminary studies on the approaches are still inconclusive.

6.1 Interfacing with Operating Systems

Most operating systems provide mechanisms for allowing concurrent execution of different processes. These mechanisms include primitives for communication and synchronization among processes. Process communication and synchronization primitives of the operating system are the basic system primitives that MDBS-II may utilize for concurrent executions of multiple requests.

6.2 Two Kinds of Interfacing Approaches

Operating systems have been characterized as either message-oriented or procedure-oriented, depending on how they implement the notions of process and synchronization [Laue79]. We could use either approach for implementing the concurrency control mechanism of MDBS-II.

Using a message-oriented operating system, there would be a fixed number of processes (one per MDBS activity). Directory management, for example, might be an activity, which could be implemented as a process. Synchronization is implemented by passing messages among processes. There is a relatively limited amount of direct sharing of data in the memory among processes. Processes for each activity are created when MDBS is started up. They are only deleted when MDBS is shut down.

Using a procedure-oriented operating system, there would be a varying number of processes (one process per user). Synchronization is implemented by direct sharing and locking of common data in the main memory. Processes are rapidly created and deleted.

In the following sections, we describe how each of the two kinds of operating system can be used for supporting concurrency control in MDBS-II. In order to simplify the discussion, we restrict the types of requests that are allowed. These restrictions mean that no changes to the directory information will be made. To show the applicability of the approaches to MDBS, we give a simplified description of the operation of MDBS using each approach. The descriptions are based on the following assumptions:

(1) There are n users.

(2) Each user has submitted one or more requests so that there are k active requests in total. The requests arrive at the controller at times t1, t2, . . . tk.

(3) Grouping of requests into transactions is not allowed.

(4) Only retrieve and update requests are allowed. Records being modified in an update request will not change cluster. Thus, there is no need for concurrency control in directory management since directory information will not change.

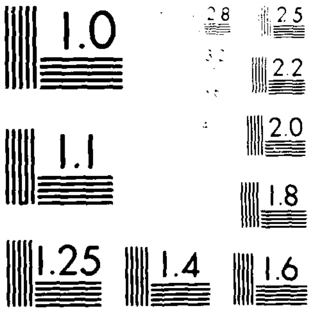
(5) Concurrency control is done at the cluster level. For example, using a procedure-oriented operating system, locking is on clusters.

(6) The scheduling of requests that reference common clusters is done using the concurrency control mechanism described in [Hsia81b].

AD-A111 981   OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/6 9/2
THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS). P--ETC(U)
JAN 82 D S KERR, A OROOJI, Z SHI   N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-82-1   NL

2 OF 2
AD A
111 981

END
DATE
FILMED
04-82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

6.2.1 Concurrency Control in MDBS-II using Message-oriented Approach

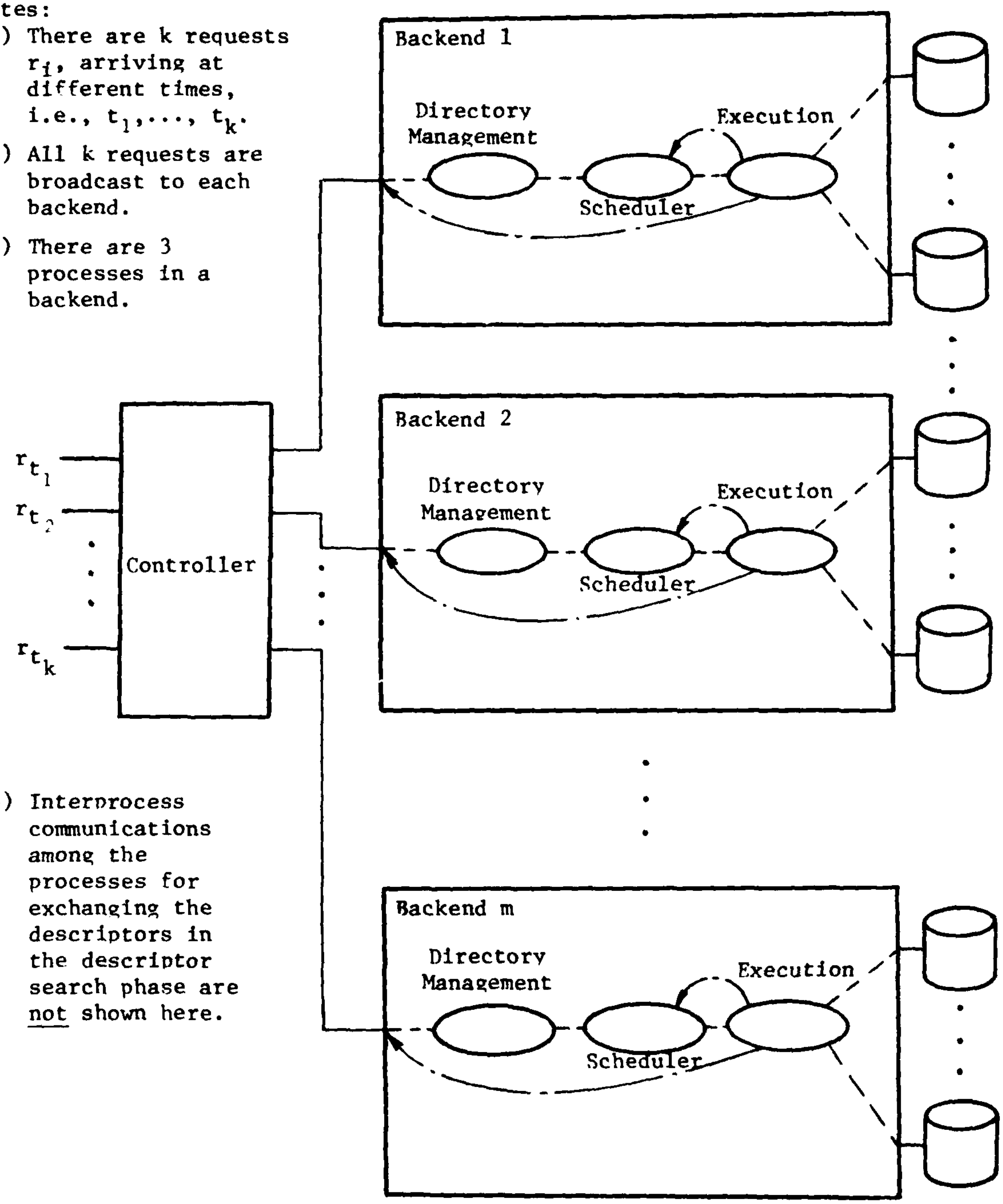
The interactions are shown in Figure 18. Requests are received at the controller and then broadcast to the backends. At each backend, a request is first input to the "directory management" process. This process determines the set of clusters needed by the request. The request and the cluster numbers of clusters determined are sent to the "scheduler" process. This process keeps a queue of requests waiting to be processed and a list of cluster numbers of clusters being accessed. This process takes a request off the queue if it can be scheduled, updates the list, and sends the request to the "request execution" process. The "request execution" process carries out the request, forwards the results to the controller, and sends a message back to the "scheduler" process indicating that the request is completed. When the "scheduler" process receives the message from the "request execution" proce^s, it updates the list, releasing all those clusters accessed by the completed request.

6.2.2 Concurrency Control in MDBS-II using Procedure-oriented Approach

The interactions are shown in Figure 19. In this approach each backend maintains a process for each active user. Thus, the number of "user" processes in MDBS-II is the product of the number of backends and the number of MDBS-II users. All "user" processes at one backend share a "cluster-lock" table. Thus, there are as many "cluster-lock" tables as there are backends.

In carrying out a user request, the "user" process at each backend consults the "cluster-lock" table at that backend. If the needed clusters are not locked, then they are locked by the process. Furthermore, the request is carried out by the process. Upon completion of the request, the process unlocks the clusters from the "cluster-lock" table. If a needed cluster is locked, then the process must wait until the cluster is unlocked. We note that there is no explicit scheduler or request queue. Instead, requests are carried out on the availability of the needed clusters as reflected from their state in the "cluster-lock" table.

Most database system implementations have used the procedure-oriented



Notes:

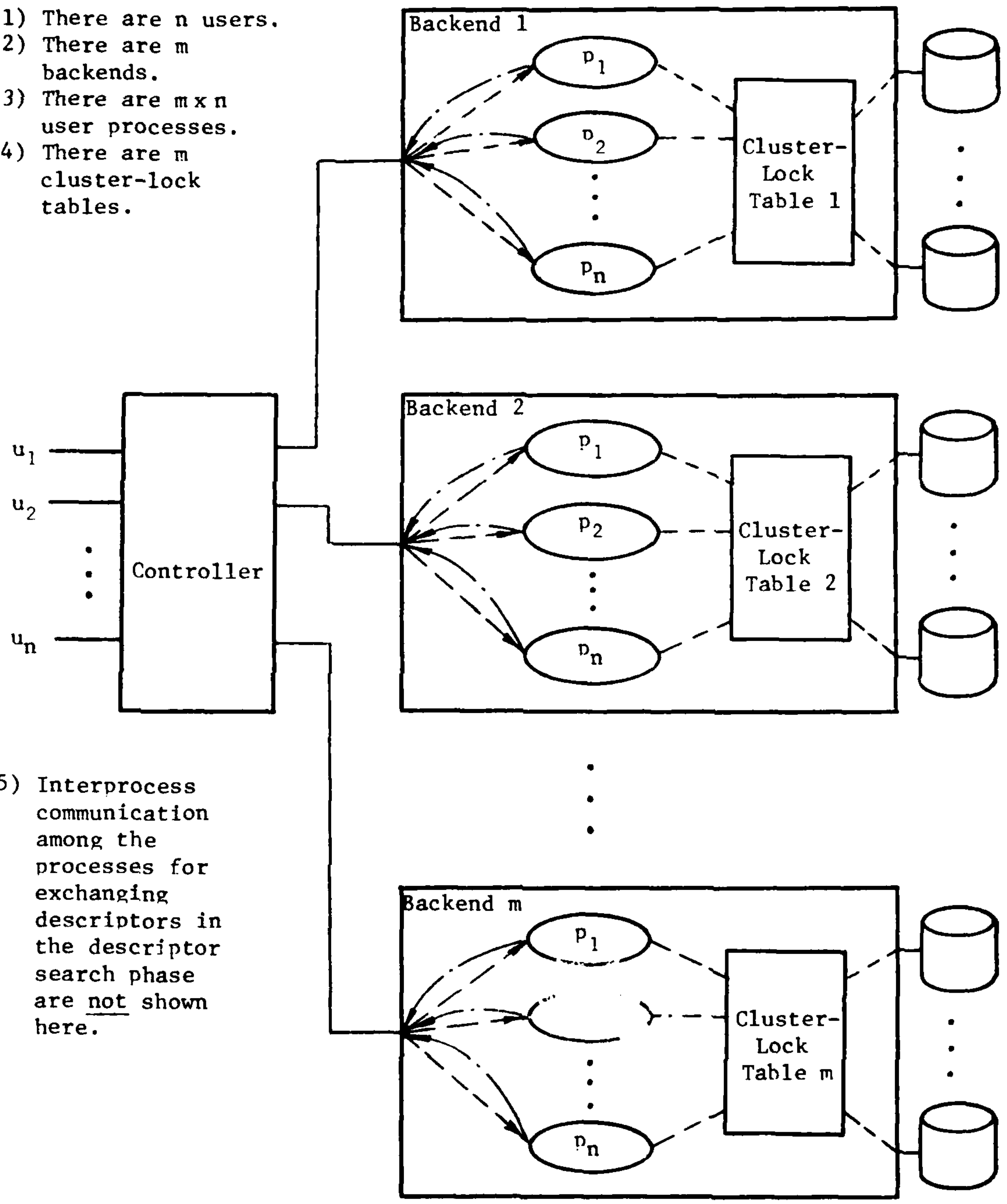
(1) There are k requests r_i, arriving at different times, i.e., $t_1, \ldots, t_k$.

(2) All k requests are broadcast to each backend.

(3) There are 3 processes in a backend.

(4) Interprocess communications among the processes for exchanging the descriptors in the descriptor search phase are <u>not</u> shown here.

Figure 18. The Message-Oriented Design for Concurrency Control in MDBS-II



Notes:
(1) There are n users.
(2) There are m backends.
(3) There are m x n user processes.
(4) There are m cluster-lock tables.

(5) Interprocess communication among the processes for exchanging descriptors in the descriptor search phase are <u>not</u> shown here.

Figure 19. The Procedure-Oriented Design for Concurrency Control In MDBS-II

approach. However, it has been suggested that a message-oriented approach might be more efficient [Ston81]. We plan to investigate both approaches more fully before choosing one for our implementation.

REFERENCES

[Astr76] Astrahan, M.M., et al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 189-222.

[Cana74] Canaday, R.H., et al., "A Back-End Computer for Database Management," Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 575-582.

[Ferr78] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, 1978.

[Gibs70] Gibson, J.C., "The Gibson Mix," IBM Technical Report, TR00.2043, June 1970.

[Gilm79] Gilmour, R.W., Business Systems Handbook: Analysis, Design, and Documentation Standards, Prentice-Hall, 1979.

[Howd80] Howden, W.E., "Functional Program Testing", IEEE Trans. on Software Engineering, Vol. 6, No. 2, March 1980, pp. 162-169.

[Hsia70] Hsiao, D.K. and Harary, F.A., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970; Corrigenda, Communications of the ACM, 13, 3, March 1970.

[Hsia81a] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)", Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus, Ohio, July 1981.

[Hsia81b] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for performance Improvement, Functionality Expansion and Capacity Growth (Part II)", Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

[Kern78] Kernighan, B.W., and Ritchie, D.M., The C Programming Language Prentice-Hall, 1978.

[Knut71] Knuth, D.E., "An Empirical Study of Fortran Programs", Software-Practice and Experience, Vol. 1, pp. 105-133.

[Laue79] Lauer, H. and Needham, R., "On the Duality of Operating System Structures," in Proc. Second International Symposium on Operating Systems, IRIA, October 1978, reprinted in Operating Systems Review, Vol. 13, No. 2, April 1979, pp. 3-19.

[Ling79] Linger, R.C., Mills, H.D., and Witt, B.I., Structured Programming - Theory and Practice, Addison-Wesley, 1979.

[Mill71] Mills, H.D., "Chief-Programmer Teams - Principles and Procedures," IBM Report FSC 71-5108, 1971.

[Ritc74] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, 17, No. 7, July 1974, pp. 365-375.

[Stone81] Stonebraker, M., "Operating System Support for Database Management," Communicatons of the ACM, Vol. 24, No. 7, July 1981, pp. 412-418.

[Wilc77] Wilcox, C.R., "MAINSAIL Language Reference Manual", SUMEX Computer Project, Stanford University Medical Center, June 1977.

[Wulf71] Wulf, W.A., Russell, D.E., and Habermann, A.N., "BLISS: A Language for Systems Programming," Communicatons of the ACM, Vol. 14, No. 12, December 1971, pp. 780-790.

[Your79] Yourdon, E., Structured Walkthroughs (2nd Edition), Prentice-Hall, 1979.

APPENDIX A
HOW TO READ AND FOLLOW THE PROGRAM SPECIFICATIONS

In Appendices B, C and D, a large number of MDBS-I programs are described and specified. These programs represent those parts of MDBS-I that have been designed and implemented at this time.

A.1 Parts within an Appendix

Each appendix begins with an introduction which outlines the major components of the design. For example, the design of test file generation, presented in Appendix B, consists of two major components: one for generating random test data strings and the other for generating realistic test data sets. Accordingly, each major component is described and specified in a seperate part of the appendix. Thus Appendix B has Part I and Part II.

A.2 The Format of a Part

In each part, we provide the following documentation elements:

(1) Title of the part,

(2) Name of the design,

(3) Name of the designer,

(4) Date the design was first submitted,

(5) Dates of design modifications,

(6) Statements of the design purpose, and of the input and output requirements,

(7) Formal specifications of the input and output, if necessary,

(8) Procedure names used in the design,

(9) Data structures used in the design,

(10) Program specification of the design.

A.3 Documentation Techniques for the Part

In the previous section, we list the various documentation elements. They are used to describe a design. Documentation elements 1 through 5 are written in English phrases. Document element 6 is written in prose. On the other hand, document elements 7 through 10 can be expressed more effectively using other means as described in Chapter 2. Specifically, we use Backus-Naur form (BNF) for writing the specifications in document element 7.

The procedure names of document element 8 are shown in a program hierarchy as discussed in Section 2.2.4 and depicted in Figure 10. The use of the hierarchy makes clear the calling sequences of the procedures named. The data structures of documentation element 9 are specified in either SSL or in the C programming language. In documentation element 10, the procedures, themselves, are specified in SSL.

Except for the programming team that writes the procedures, other teams will usually not be interested in the internal logic of the procedures. Consequently, they need only know the higher-level specifications of the procedures. SSL as described in Section 2.2.2 and depicted in Figure 9 is an ideal specification language for revealing the design of the procedures from a top-to-bottom-and-layer-to-layer way. It also works well with the hierarchical organization of procedures.

APPENDIX B

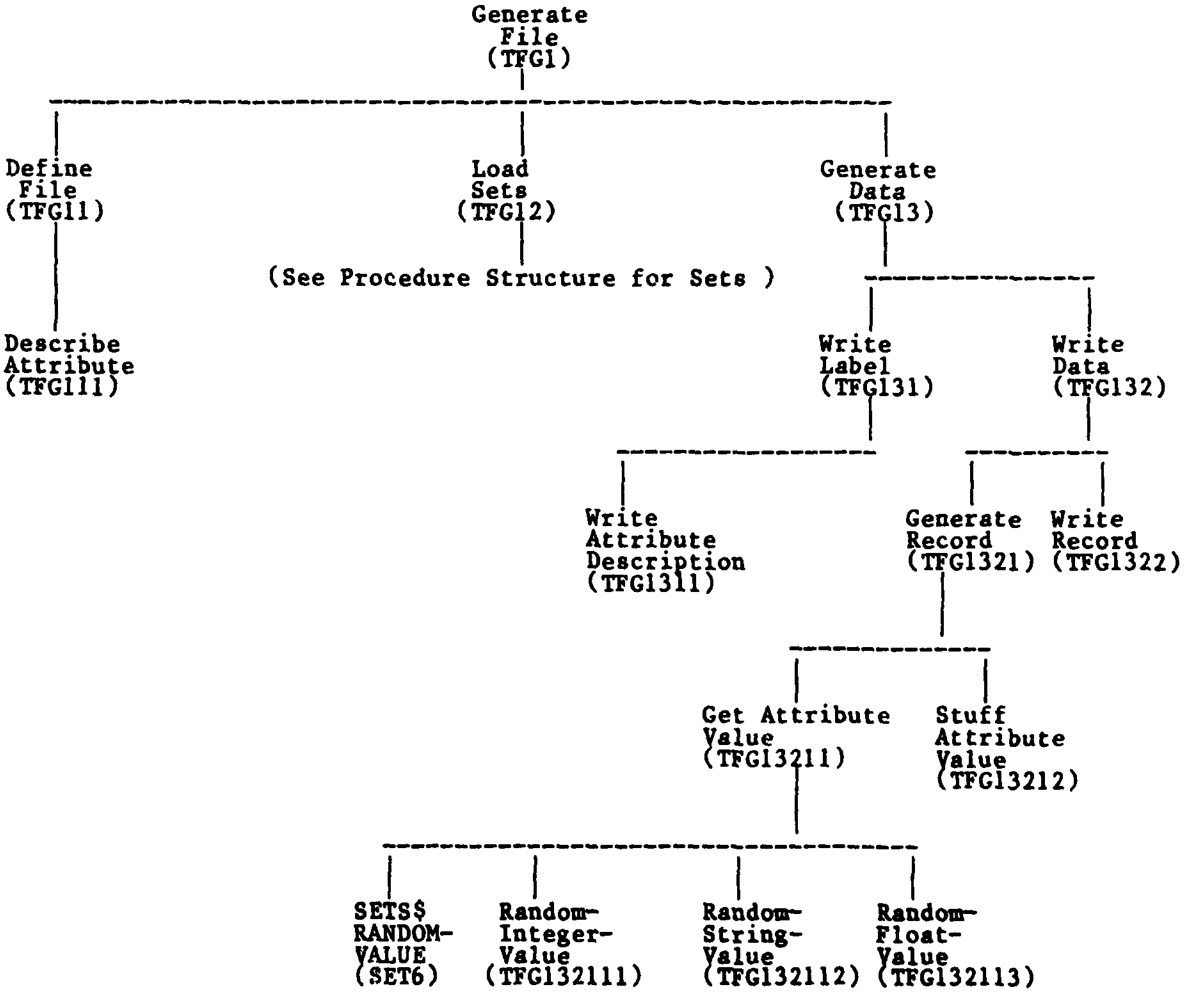
THE SSL SPECIFICATION FOR TEST FILE GENERATION

The program specification for test file generation is shown in this appendix. The specification design is composed of two parts. In part two, all procedures and data structures that are required to define set members and then to select a particular member for a value in a record are specified. In part one, all the procedures that are not concerned with data sets are specified.

B.1 Part I - Generating Random Test Data Strings

```
/*    (1) Part I - Generating Random Test Data Strings                       */
/*    (2) Design:GENERATE FILE(FILE-NAME)                                    */
/*    (3) Designer: D.S. Kerr                                                */
/*    (4) Date: July 23, 1981                                                */
/*    (5) Modified July 30, 1981                                             */
/*              August 4, 1981                                               */
/*              August 11, 1981 - removed SETS to Part II, no other changes  */
/*              August 25, 1981 - changed identification of set from         */
/*                  SetNumber to SetPointer in AttributeDescription          */
/*              January 3, 1982 - description changed, no changes to the     */
/*                  design itself                                            */
/*                                                                           */
/*     (6) Purpose:                                                          */
/*    The purpose of this system is to generate a file of test data which    */
/* can be applied to MDBS. The user specifies the FILE-NAME,                 */
/* NUMBER-OF-RECORDS, and  the NUMBER-OF-ATTRIBUTES-PER-RECORD. The user     */
/* then specifies how the values of each attribute are to be chosen:         */
/* randomly from a predefined set, randomly from a range of integers, or     */
/* as a random character string.                                             */
/*  A set is characterized by a set-name, type(length), number-of-members,   */
/* and the members. It will be stored in a file called set-name in a         */
/* library of sets.                                                          */
/*                                                                           */
/*     A distribution function, UNIFORM(min,max), must be provided. It       */
/* should generate a random integer between min and max.                     */
/*                                                                           */
/*      (7) Output:                                                          */
/*     Output is a file of records where each record has the form            */
/*               field1$field2$ . . . $fieldn#                               */
/* The actual data output is a character string. $ is a special character    */
/* to seperate fields, # is a special character to seperate records. A       */
/* more formal definition is given below.                                    */
/*                                                                           */
/* Notation  {item} ... means 1 or more occurrences of item                  */
/*           [item] ... means 0 or more occurrences of item                  */
/*                                                                           */
/* file ::= label data                                                       */
/* label ::= number-of-records $ number-of-attributes $ attribute-body        */
/* number-of-records ::= integer     the number of records in the file       */
/* number-of-attributes ::= integer  the number per file                     */
/*                                                                           */
/* attribute-body ::= {attribute-description $ } ...                          */
/* attribute-description ::= type(length) source-domain $ distribution        */
/* type ::= INTEGER | STRING | FLOAT | others to be added later               */
/* source-domain ::= set-name | RANDOM-INTEGER | RANDOM-STRING                */
/* distribution ::= distribution-function(parameter-list)                     */
/* distribution-function ::= UNIFORM | others to be added later               */
/* parameter-list ::= integer [,integer] ...                                  */
/* length ::= integer                in bytes of an attribute                 */
/* integer ::= {digit} ...           usual definition                         */
/* set-name ::= filename             sets are described below                 */
/* filename ::=   any legal RSX ( UNIX) filename                              */
/*                                                                            */
/* data ::= {record#} ...                                                     */
/* record ::= field [$field] ...                                              */
/* field ::= string                  actual field value                       */
/* string ::= string character | character                                    */
/* character ::= digit | letter | specialcharacter                            */
/* digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9                            */
/* letter ::= lowercaseletter | uppercaseletter                               */
/* lowercaseletter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z    */
/* uppercaseletter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z    */
/* specialcharacter ::= ! | @ | # | $ | % | ^ | & | * | ( | ) | | | + | ~ |   */
/*                      ` | = | - | { | } | ] | [ | : | " | ' | ; | < | > |   */
/*                      , | . | ? | / | ^                                     */
/*                                                                            */
```

(8) Procedure Hierarchy for File Generation


Generate File (TFG1)
Define File (TFG11)
Load Sets (TFG12)
Generate Data (TFG13)
Describe Attribute (TFG111)
(See Procedure Structure for Sets)
Write Label (TFG131)
Write Data (TFG132)
Write Attribute Description (TFG1311)
Generate Record (TFG1321)
Write Record (TFG1322)
Get Attribute Value (TFG13211)
Stuff Attribute Value (TFG13212)
SETS$ RANDOM-VALUE (SET6)
Random-Integer-Value (TFG132111)
Random-String-Value (TFG132112)
Random-Float-Value (TFG132113)

(9) Data Structures

```
#define FILENAMELENGTH 10 /* maximum number of characters in a file name   */
#define MAXATTRIBUTES  10 /* maximum number of attributes in a record      */
#define MAXPARAMS      10 /* maximum number of parameters for distribution */
                          /* functions                                     */

    struct AttributeDescription {
        char Type; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
        int  Length; /* number of bytes */
        int  Source;/* SET = 1 , RANDOM_INTEGER = 2 , RANDOM_STRING = 3 */
        char SetName[ FILENAMELENGTH ]; /* defined only for sets */
**      struct SetDef *SetPointer; /* to the set definition     */
        int  Distribution; /* UNIFORM = 1 , ... */
        int  Parameters[MAXPARAMS]; /* of the distribution */
        }

    struct FileDescription {
        char FileName[ FILENAMELENGTH ]; /* name of the file */
        int  NumberOfRecords; /* to be generated in the file */
        int  NumberOfAttributes; /* in a record */
        struct AttributeDescription Description[MAXATTRIBUTES];
        }
```

(10) Program Specifications

```
1. job GENERATE_FILE;
        /* Uses: DEFINE_FILE - to fill in the attribute descriptions    */
        /*       LOAD_SETS - to load the sets                           */
        /*       GENERATE_DATA - to generate the actual data            */
2.     perform DEFINE_FILE;
3.     perform LOAD_SETS;
4.     perform GENERATE_DATA;
5. end job

2.1. proc DEFINE_FILE; /* fill in file description */
2.2       read and store FileName, NumberOfRecords, NumberOfAttributes;
2.3       NumberOfAttributes <= MAXATTRIBUTES;

          /* Uses DESCRIBE_ATTRIBUTE( pointer_to_description ) - to read  */
                  and store an attribute description                      */

2.4       int i; /* attribute subscript */

2.5       for i from 1 to NumberOfAttributes do
2.6           perform DESCRIBE_ATTRIBUTE( &FileDescription.Description[i]);
2.7       end for
2.8     end proc

4.1 proc GENERATE_DATA;
        /* Uses the information in the file and attribute descriptions to  */
        /* generate the file.                                               */

4.2     open file(file_name);
4.3     perform WRITE_LABEL;
4.4     perform WRITE_DATA;
4.5     close file(file_name);
4.6  end proc

4.3.1   proc WRITE_LABEL;
                /* Uses the information in the file and attribute   */
                /* descriptions to write the label.                 */

4.3.2       int i; /* attribute subscript */
4.3.3       $char is the special character used to seperate fields

4.3.4       write NumberOfRecords, $char, NumberOfAttributes, $char;
4.3.5       for i from 1 to NumberOfAttributes do
4.3.6           perform WRITE_ATTRIBUTE_DESCRIPTION(
4.3.7                   &FileDescription.Description[ i ] );
4.3.8           write $char;
4.3.9       end for
4.3.10  end proc
```

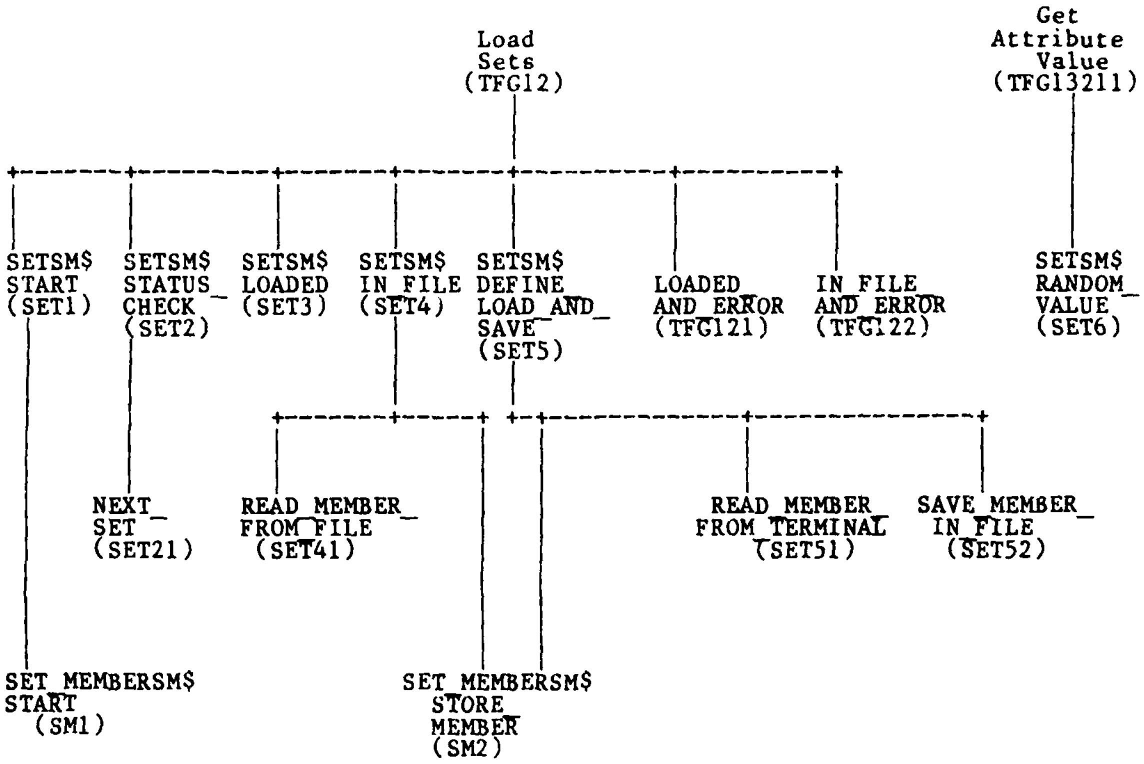
```
4.4.1   proc WRITE_DATA;
                /* Uses the information in the file and attribute descriptions
  */
                /* to generate and write the records.                      */

                Uses:   GENERATE_RECORD( record ) returns record as a
                                character string
                        WRITE_RECORD( record ) where record is the character
                                string to be added to the output file.

4.4.2           int rec_no; /* index for records */
4.4.3           #char is the special character used to seperate records

4.4.3       for rec_no from 1 to number_of_records
4.4.4       do
4.4.5               perform GENERATE_RECORD( record );
4.4.6               perform WRITE_RECORD( file_name, record );
4.4.7               Write #char;
4.4.8       end for;
4.4.9   end proc

4.5.1   proc GENERATE_RECORD(record);
                output: record - character string

                Uses:   GET_ATTRIBUTE_VALUE( pointer_to_description ) -
                                to determine a value (character string ) for
                                this attribute
                        STUFF_ATTRIBUTE_VALUE( value, record ) - appends
                                value to record.

4.5.2       int attr_no; /* index for attributes */
4.5.3       string value;  /* for a particular attribute, in string form */

4.5.4       for attr_no from 1 to number_of attributes do
4.5.5           value := GET_ATTRIBUTE_VALUE(
                     FileDescription.Description[ attr_no ] );
4.5.6           perform STUFF_ATTRIBUTE_VALUE(value,record);
4.5.7        end for;
4.5.8        return(record);
4.5.9   end proc

/* end scope of FileDescription */


2.6.1 proc DESCRIBE_ATTRIBUTE( DescriptionPtr );
            input: DescriptionPtr - /* points to AttributeDescription */

2.6.2   struct AttributeDescription {
2.6.3           char Type; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
2.6.4           int  Length; /* number of bytes in string to be generated */
2.6.5           int  Source;/* SET = 1 , RANDOM_INTEGER = 2 , RANDOM_STRING = 3
 */
2.6.6           char SetName[ FILENAMELENGTH ]; /* defined only for sets */
2.6.7**         struct SetDef *SetPointer; /* to the set definition     */
2.6.8           int  Distribution; /* UNIFORM = 1 , ... */
2.6.9           int  Parameters[MAXPARAMS]; /* of the distribution */
2.6.10          }

        /* enter attribute into attribute table */

2.6.11  define Type;
2.6.12  define Length;
2.6.13  define Source;
2.6.14    if Source is SET
2.6.15        then
2.6.16            define SetName;
2.6.17  define Distribution;
2.6.18  define Parameters;
2.6.19 end proc
```

```
4.3.6.1 proc WRITE_ATTRIBUTE_DESCRIPTION( DescriptionPtr );
        /* Writes the description of one attribute into the output file  */

4.3.6.2     $char is the special character used to seperate fields

4.3.6.3     Write the description;
4.3.6.4 end proc


4.5.5.1 proc GET_ATTRIBUTE_VALUE( DescriptonPtr );
                input:  DescriptionPtr /* points to AttributeDescription */
                returns: value character string

                Uses:   SETS$RANDOM_VALUE( DescriptionPtr )
                        RANDOM_INTEGER_VALUE( DescriptionPtr )
                        RANDOM_STRING_VALUE( DescriptionPtr )
                        RANDOM_FLOAT_VALUE( DescriptionPtr )
                                each returns a character string representation
                                of the appropriate value

4.5.5.2 struct AttributeDescription {
4.5.5.3     char Type; /* INTEGER = ´i´ , STRING = ´s´ , FLOAT = ´f´ */
4.5.5.4     int  Length; /* number of bytes */
4.5.5.5     int  Source;/* SET = 1 , RANDOM_INTEGER = 2 , RANDOM_STRING = 3 */
4.5.5.6     char SetName[ FILENAMELENGTH ]; /* defined only for sets */
4.5.5.7**           struct SetDef *SetPointer; /* to the set definition     */
4.5.5.8     int  Distribution; /* UNIFORM = 1 , ... */
4.5.5.8     int  Parameters[MAXPARAMS]; /* of the distribution */
4.5.5.10    }
            /* get a value for attribute attr_no */

4.5.5.11     case DescriptionPtr.Source value
4.5.5.12        ´set´  : value :=
                    SETS$RANDOM_VALUE( DescriptionPtr );

4.5.5.13        ´integer´ : value :=
                    RANDOM_INTEGER_VALUE( DescriptionPtr );

4.5.5.14        ´string´ : value :=
                    RANDOM_STRING_VALUE( DescriptionPtr );

4.5.5.15        ´float   : value :=
                    RANDOM_FLOAT_VALUE( DescriptionPtr );
4.5.5.16     end case
4.5.5.17     return(value);
4.5.5.18  end proc


4.5.6.1 proc STUFF_ATTRIBUTE_VALUE(value,record);
        input:  value - character string
        input/output: record - character string
4.5.6.2     /* puts value into record */
4.5.6.3 end proc


4.4.6.1 proc WRITE_RECORD( file_name, record );
        input: file_name, record
        /* actually writes record to file file_name  */
4.4.6.2 end proc

4.5.5.13.1 proc RANDOM_INTEGER_VALUE( DescriptionPtr );
             input: DescriptionPtr
             /* returns an integer value as a character string */
4.5.5.13.2 end proc

4.5.5.14.1 proc RANDOM_STRING_VALUE( DescriptionPtr );
              input: DescriptionPtr
              /* returns a character string value  */
4.5.5.14.2 end proc

4.5.5.15.1 proc RANDOM_FLOAT_VALUE( DescriptionPtr );
              input: DescriptionPtr
              /* returns a floating point number as a character string */
4.5.5.15.2 end proc
```

B.2 Part II - Generating Realistic Test Data Sets

```
/*      (1) Part II - Generating Realistic Test Data Sets                    */
/*      (2) Design: SETS in File Generation                                  */
/*      (3) Designer: D. S. Kerr                                             */
/*      (4) Date: July 23, 1981                                              */
/*      (5) Modified: July 30, 1981                                          */
/*                August 4, 1981                                             */
/*                August 24, 1981                                            */
/*                August 27, 1981 Minor changes after final walkthrough      */
/*                                                                           */
/* (6) Purpose:                                                              */
/* A set is characterized by a set_name, type, length, NumberOfMembers,      */
/* and the members.  It will be stored in a file called set_name in a        */
/* library of sets.                                                          */
/*                                                                           */
/* (7) Input and Output Data                                                 */
/* The form of the data in a file is shown below.  $ is a special character */
/* used to seperate fields in the file label.  # is a special character      */
/* used to seperate members.                                                 */
/*                                                                           */
/* set_file_name ::=   any legal RSX ( UNIX) filename                        */
/* set_file ::= set_label set_data                                           */
/* set_label ::= DataType $ DataLength $ NumberOfMembers $                   */
/* set_data ::= { Member # } ...                                             */
/* Member ::= { character } ...                                              */


/* DataType ::= INTEGER | STRING | FLOAT | others to be added later          */
/* DataLength ::= integer           in bytes, of a set member                */
/* NumberOfMembers ::= integer      the number of members in the set         */
/* integer ::= { digit } ...        usual definition                         */
/*                                                                           */
/* character ::= digit | letter | specialcharacter                           */
/* digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9                           */
/* letter ::= lowercaseletter | uppercaseletter                              */
/* lowercaseletter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z    */
/* uppercaseletter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z    */
/* specialcharacter ::= ! | @ | # | $ | % |   | & | * | ( | ) | _ | + | ~ |   */
/*                      ` | = | - | { | } | ] | [ | : | " | ' | ; | < | > |   */
/*                      , | . | ? | / | \|&                                  */
/*                                                                           */
```

(8) Procedure Structure for Sets


Load Sets (TFG12)
Get Attribute Value (TFG13211)
SETSM$ START (SET1)
SETSM$ STATUS_ CHECK (SET2)
SETSM$ LOADED (SET3)
SETSM$ IN FILE (SET4)
SETSM$ DEFINE LOAD AND_ SAVE (SET5)
LOADED AND ERROR (TFG121)
IN FILE AND ERROR (TFG122)
SETSM$ RANDOM_ VALUE (SET6)
NEXT_ SET (SET21)
READ MEMBER_ FROM FILE (SET41)
READ MEMBER FROM TERMINAL (SET51)
SAVE MEMBER_ IN FILE (SET52)
SET MEMBERSM$ START (SM1)
SET MEMBERSM$ STORE MEMBER (SM2)

(10) Program Specifications

```
3.1      proc LOAD_SETS;
             /* For each set to be loaded in main memory, fill in SetDefinition.    */
             /* Also fill in SetPointer in AttributeDescription.  Set may already   */
             /* be loaded for a previous attribute.  It may also be in a previously */
             /* defined library of sets.  If it is not already defined, then it must */
             /* be read from a terminal.                                             */
             /* Any new sets defined will be added to this library.                 */

             Uses:
                 SETSM$START - to initialize SETSM module
                 SETSM$STATUS_CHECK( NamePtr, Type, Length, Status, SetPtr )
                  - Returns the status of set 'NamePtr'.  Also returns SetPtr, a
                  pointer to a structure of type SetDef if there were no errors.
                  Possible values of Status are:
                   LOADED - Already loaded in primary memory for a previous attribute
                   LOADED_AND_ERROR - Loaded but set description and attribute
                          description do not match
                   IN_FILE - Already defined in a file but not yet loaded in primary
                          memory
                   IN_FILE_AND_ERROR - Defined but set description and attribute
                          description do not match
                   NEW - Set not yet defined
                 SETSM$LOADED( NamePtr, SetPtr ) - Returns SetPtr for set 'NamePtr'
                          which is already loaded in primary memory.
                 SETSM$IN_FILE( NamePtr, SetPtr ) - Loads set 'NamePtr' from file
                          into primary memory.  Returns the corresponding SetPtr.
                 SETSM$DEFINE_LOAD_AND_SAVE( NamePtr, Type, Length, SetPtr ) - When
                          set has not previously been defined.  Reads the set
                          from the terminal, loads it into primary memory, and
                          saves it in a file for future use. SetPtr is returned.

                 LOADED_AND_ERROR - Used to fix error.
                 IN_FILE_AND_ERROR - Used to fix error.

3.2      int Status; /* of a set takes on values shown above */
3.3      int i; /* attribute subscript */
3.4      char *CurrentAttributePtr; /* pointer to the current attribute if a set.*/
3.5      set description *SetPtr; /* pointer to the set description */

3.6              perform SETSM$START;
3.7              for i from 1 to NumberOfAttributes do
3.8                  if FileDescription.Description[i].Source is SET
3.9                  then /* then there are 3 cases */
                          /* 1. set already loaded in memory for a previous attribute*/
                          /* 2. set already defined in a file  */
                          /* 3. set must be read from terminal, loaded and saved */
3.10                         CurrentAttributePtr = &FileDescription.Description[i];
3.11                         perform SETSM$STATUS_CHECK( CurrentAttributePtr->SetName,
                                     CurrentAttributePtr->Type,
                                     CurrentAttributePtr->Length,
                                     Status, SetPtr );
3.12                         case Status value
3.13                             LOADED: perform SETSM$LOADED( SetPtr );
3.14                             IN_FILE: perform SETSM$IN_FILE( SetPtr );
3.15                             NEW: perform SETSM$DEFINE_LOAD_AND_SAVE( SetPtr );
3.16                             LOADED_AND_ERROR: perform LOADED_AND_ERROR;
3.17                             IN_FILE_AND_ERROR: perform IN_FILE_AND_ERROR;
3.18                         end case;

3.19                 CurrentAttributePtr->SetPointer = SetPtr;
3.20                 end if;
3.21             end for;
3.22         end proc

3.16.1   proc LOADED_AND_ERROR;
             /* May ask for redefinition of set name or attribute description.  */
3.16.2   end proc

3.17.1   proc IN_FILE_AND_ERROR;
             /* May ask for redefinition of set name or attribute description.  */
3.17.2   end proc
```

```
        module SETSM;
                /* Values of defined constants must be determined before system   */
                /* is fully operational.                                          */
        #define FILENAMELENGTH 10 /* maximum number of characters in a file name   */
        #define MAXSETS        10 /* maximum number of sets allowed               */
        #define MAXMEMBERS     10 /* maximum number of members in a set           */
        #define MAXSETSTORAGE 10 /* maximum storage to hold the sets              */

            struct SetDef {
                char Name[ FILENAMELENGTH];
                char DataType; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
                int  DataLength; /* number of bytes */
                int  NumberOfMembers; /* in the set.  When set is being stored    */
                                    /* is the number of members currently in the set. */
                char *MemberPtr[MAXMEMBERS]; /* MemberPtr[i] points to the      */
                        /* character string value of the i-th member of this set. */
                }
            struct SetDef SetDefinition[MAXSETS]; /* one for each set */

            struct SetDef *SetAvailablePtr; /* points to next available set */

            exported: START, STATUS_CHECK, LOADED, IN_FILE, DEFINE_LOAD_AND_SAVE,
                RANDOM_VALUE

        internal: READ_MEMBER_FROM_FILE, READ_MEMBER_FROM_TERMINAL,
                    SAVE_MEMBER_IN_FILE

        uses START and STORE_MEMBER from SET_MEMBERSM module


1.1     proc START;
            /* initializes SetAvailablePtr and SET_MEMBERSM module */
1.2         Initialize SetAvailablePtr to initial SetDefinition.
1.3         perform SET_MEMBERSM$START;
1.4     end proc
```

```
2.1     proc STATUS_CHECK( input : SetNamePtr, Type, Length,
                          output : Status, SetPtr );
                input:  SetNamePtr /* of set to be checked        */
                        Type       /* from attribute description */
                        Length     /* from attribute description */

                output: Status /* of set 'NamePtr'.  Possible values are:
                           LOADED - Already loaded in primary memory
                           LOADED_AND_ERROR - loaded but descriptions do not match
                           IN_FILE - Already defined in a file
                           IN_FILE_AND_ERROR - Defined but descriptions don't match
                           NEW - not yet defined */

                        SetPtr   /* Pointer to structure of type SetDef.  This set
                                    /* is to be defined if there are no errors.   */

        /* SetName pointed to by SetNamePtr is the same as the name of the file   */
        /* which holds the set.                                                    */

2.2         if there exists a set j, such that SetDefinition[j].Name matches
                        set name identified by SetNamePtr
2.3             then /* Check set description and attribute description */
2.4                 if descriptions match
2.5                     then
2.6                         Status = LOADED;
2.7                         define SetPtr to point to SetDefinition[j];
2.8                     else  Status = LOADED_AND_ERROR;
2.9                 end if;

2.10            else /* There are still two possibilities.  First, check for set */
                     /* in library of sets */
2.11                open file( SetNamePtr );
2.12                if open successful
2.13                    then /* check set description and attribute description */
2.14                        read set description from file;
2.15                        if descriptions match
2.16                            then
2.17                                Status = IN_FILE;
2.18                                perform NEXT_SET( SetPtr );
2.19                                store set name, type, length & NumberOfMembers
2.20                                        in set description;
2.21                            else  Status = IN_FILE_AND_ERROR;
2.22                        end if;

2.23                    else /* Since open was not successful, set must not have */
2.24                         /* been previously defined */
2.25                        Status = NEW;
2.26                        perform NEXT_SET( SetPtr );
2.27                        store set name, type and length in set description;
                                /* Note that NumberOfMembers is not known until the */
                                /* set has been read from the terminal.             */
2.28                end if;
2.29            end if;
2.30    end proc;

        internal procedure which requires access to all of SetDefinition

2.18.1  proc NEXT_SET( output : SetPtr );
            /* Returns a pointer to the next available set.  Increments    */
            /* SetAvailablePtr.                                             */

            /* SetPtr and SetAvailablePtr -  pointers to structures of type SetDef. */

2.18.2      SetPtr = SetAvailablePtr;
2.18.3      Increment SetAvailablePtr;

2.18.4  end proc

        end_scope_of SetDefinition

3.1     proc LOADED( input : SetPtr );
            /* actually does not have to do anything */
            input:  SetPtr /* a pointer to a structure of type SetDef */
3.2     end proc
```

```
4.1     proc IN_FILE( input : SetPtr );
              input: SetPtr /* a pointer to a structure of type SetDef, the   */
                          /* set to be input */
                 /*    Reads in the actual set members from the file and    */
                 /*    stores them in set SetPtr.                           */

4.2        struct SetDef {
4.3           char Name[ FILENAMELENGTH];
4.4           char DataType; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
4.5           int  DataLength; /* number of bytes */
4.6           int  NumberOfMembers; /* in the set.  When set is being stored    */
                            /* is the number of members currently in the set. */
4.7           char *MemberPtr[MAXMEMBERS]; /* MemberPtr[i] points to the      */
                      /* character string value of the i-th member of this set. */
4.8           }

4.9        int i; /* index for set members */

4.10       /* read and store the members */
4.11       for i from 1 to NumberOfMembers do
4.12          perform READ_MEMBER_FROM_FILE( SetPtr->Name , MemberValue );
4.13          perform SET_MEMBERSM$STORE_MEMBER( input : MemberValue,
                    output : NewMemberPtr );
4.14          MemberPtr[ i ] = NewMemberPtr;

4.15       end for;
4.16       close file( SetPtr->Name );

4.17    end proc


5.1     proc DEFINE_LOAD_AND_SAVE( input : SetPtr );
           input:  SetPtr /* pointer to structure of type SetDef */

           /* Set has not previously been defined.  Reads the set from the      */
           /* terminal, loads it into primary memory, and saves it in a file for */
           /* future use.  Stores NumberOfMembers in set description.           */

5.2        struct SetDef {
5.3           char Name[ FILENAMELENGTH];
5.4           char DataType; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
5.5           int  DataLength; /* number of bytes */
5.6           int  NumberOfMembers; /* in the set.  When set is being stored    */
                            /* is the number of members currently in the set. */
5.7           char *MemberPtr[MAXMEMBERS]; /* MemberPtr[i] points to the      */
                      /* character string value of the i-th member of this set. */
5.8           }

5.9     int i; /* index */
5.10    SetNamePtr /* pointer to the name of the set */

        /* Define and load set into primary memory. */
5.11      i = 0;
5.12      perform READ_MEMBER_FROM_TERMINAL( MemberValue );
5.13      while ( MoreMembers ) do
5.14         if i > MAXMEMBERS
5.15            then perform ErrorRoutine;
5.16         perform SET_MEMBERSM$STORE_MEMBER( input : MemberValue,
                   output : NewMemberPtr , ErrorStatus );
5.17         if ErrorStatus = NO_SPACE
5.18            then perform ErrorRoutine;
5.19         increment i;
5.20         MemberPtr[ i ] = NewMemberPtr;
5.21         perform READ_MEMBER_FROM_TERMINAL( MemberValue );
5.22      end while;
5.23      store i in NumberOfMembers in SetDef;

5.24    /* Save set in file. */
5.25       SetNamePtr = SetPtr->Name;
5.26       open file( SetNamePtr );
5.27       write set description to file;
           /* write set members to file */
5.28          for i from 1 to NumberOfMembers do
5.29             perform SAVE_MEMBER_IN_FILE( SetNamePtr,
                          SetPtr->MemberPtr[ i ] );
5.30          end for;
5.31       close file( SetNamePtr );

5.32    end proc
```

```
6.1      proc RANDOM_VALUE( input : AttributeDescriptionPtr );
             / *returns pointer to value */
             input: AttributeDescriptionPtr /* pointer to structure of type  */
                    /* AttributeDescription                                 */

6.2          struct AttributeDescription {
6.3              char Type; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
6.4              int  Length; /* number of bytes */
6.5              int  Source;/* SET = 1 , RANDOMINTEGER = 2 , RANDOMSTRING = 3 */
6.6                char SetName[ FILENAMELENGTH ]; /* defined only for sets */
6.7                struct SetDef *SetPointer; /* to the set definition */
6.8              int  Distribution; /* UNIFORM = 1 , ... */
6.9              int  Parameters[MAXPARAMS]; /* of the distribution */
6.10             }

6.11         struct SetDef {
6.12             char Name[ FILENAMELENGTH];
6.13             char DataType; /* INTEGER = 'i' , STRING = 's' , FLOAT = 'f' */
6.14             int  DataLength; /* number of bytes */
6.15             int  NumberOfMembers; /* in the set.  When set is being stored    */
                                 /* is the number of members currently in the set. */
6.16             char *MemberPtr[MAXMEMBERS]; /* MemberPtr[i] points to the     */
                         /* character string value of the i-th member of this set. */
6.17             }

             /* Uses SetPointer, Distribution, Parameters                      */
             /* to find the value of a random member of the set          */
             /* Note that different attributes may use the same set, but */
             /* with different distribution functions.                    */
6.18     end proc


         end_of_exported_procedures


         start_of_internal_procedures

4.12.1   proc READ_MEMBER_FROM_FILE( input : SetNamePtr,
                                     output : MemberValue );
             /* Reads a member from file SetNamePtr */
4.12.2   end proc;

5.12.1   proc READ_MEMBER_FROM_TERMINAL( output : MemberValue );
             /* Reads a member from terminal */
             /* Uses NULL to indicate no more members. */
5.12.2   end proc;

5.29.1   proc SAVE_MEMBER_IN_FILE( input : SetNamePtr, MemberPtr );
             input: SetNamePtr
                    MemberPtr
             /* writes member into file SetNamePtr */
5.29.2   end proc;


         end_of_internal_procedures

         end module SETSM
```

```
        module SET_MEMBERSM
            char SetData[MAXSETSTORAGE]; /* holds the set members - pointed to  */
                                         /* by MemberPtr                        */

            char *MemberAvailablePtr; /* Points to 1st available space in       */
                                      /* SetData        */

        /* first character of member m of set s is SetDefinition[s].MemberPtr[m] */
        /* last character is SetDefinition[s].MemberPtr[m+DataLength-1]          */
        /*     SetDefinition[s].MemberPtr[m+DataLength] is NULL ( = '\0')        */

1.1     proc START;
            /* Initialize MemberAvailablePtr to beginning of SetData.            */
1.2     end proc

2.1     proc STORE_MEMBER( input :MemberValue,
                           output : NewMemberPtr , ErrorStatus );

            /* MemberValue is the value to be stored.  It is stored in the next  */
            /* available spaces in SetData.  NewMemberPtr is returned after it   */
            /* is set to point to this value.  MemberAvailablePtr is incremented.*/
            /* ErrorStatus is set to OK.  If there is no room, then ErrorStatus  */
            /* is set to NO_SPACE.                                               */

            /* It should be noted that only STORE_MEMBER requires access to all  */
            /* of SetData.  All other routines get pointers to a particular      */
            /* value in SetValue and use only that particular value.             */

            /* SetData looks like                                                */
            /*                                                                   */
            /*  ------------------------------------------------------------    */
            /*  | value NULL value NULL . . . last-value NULL      unused   |    */
            /*  ------------------------------------------------------------    */
            /*                                                ^                 */
            /*                                                |                 */
            /* where MemberAvailablePtr points to the first available space.    */

2.2         if there is not enough space
2.3             then ErrorStatus = NO_SPACE;
2.4             else
2.5                 ErrorStatus = OK;
2.6                 Store MemberValue in SetData from MemberAvailablePtr on;
2.7                 NewMemberPtr = MemberAvailablePtr;
2.8                 increment MemberAvailablePtr;
2.9         end if

2.10    end proc


        end module SET_MEMBERSM
```

APPENDIX C

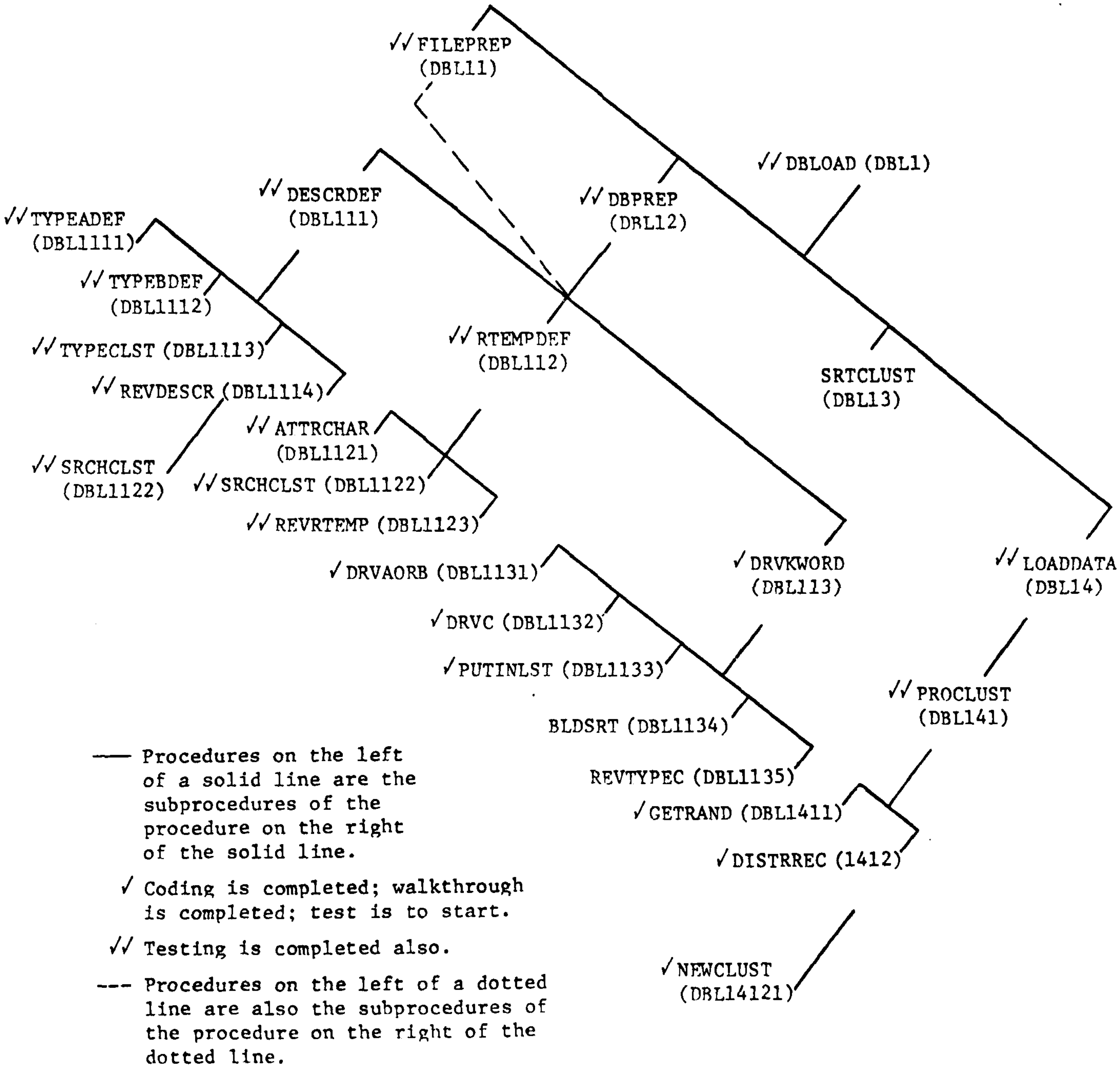
THE SSL SPECIFICATION FOR DATABASE LOAD

The program specification for database load is shown in this appendix. The specification design is composed of two parts. Part one includes all procedures for the database load subsystem. Part two includes the specifications for the Record Template module.

C.1 Part I - Database Load Subsystem

```
/*    (1) Part I - Database Load Subsystem            */
/*    (2) Design:    DATABASE LOAD UTILITY            */
/*    (3) Designer:  P. R. Strawser                   */
/*    (4) Date:      August 25, 1981                  */
/*    (5) Modified:  September 16, 1981               */
/*                   Changes marked with '**' in SSL. */
/*                                                    */
/*    (6) Purpose:                                    */
/*        The database load utility gives MDBS users  */
/*        the capability to load pre-existing data from */
/*        other database systems or other files into  */
/*        the MDBS system.  This utility is designed to */
/*        run on the MDBS controller.                 */
/*        The files being loaded are assumed to be of */
/*        fixed length records, all records in a file */
/*        having the same format.  The files are also */
/*        assumed to be resident at the controller.   */
/*        The database load utility, using these files */
/*        and other information supplied by the user, */
/*        constructs the DDIT and AT tables required by */
/*        the directory management subsystem.  It also */
/*        formats the input records as required for   */
/*        storage in the database, organizes the records*/
/*        into clusters, and distributes the records and*/
/*        the directory management data to the backends.*/
/*                                                    */
/*    (7) Output:                                     */
/*        Output is DDIT and AT information for the   */
/*        directory management subsystem, and records */
/*        formatted for storage at the backends.      */
```

(8) Procedure Hierarchy for Database Load



—— Procedures on the left of a solid line are the subprocedures of the procedure on the right of the solid line.

√ Coding is completed; walkthrough is completed; test is to start.

√√ Testing is completed also.

--- Procedures on the left of a dotted line are also the subprocedures of the procedure on the right of the dotted line.

(9) Data Structures

Data structure definitions are included at the beginning of each procedure definition in (10) below.

(10) Program Specifications

First Level Specifications for Database Load

```
subsystem  DATABASE_LOAD;       /* DBLOAD (DBL1) */
/* Prepare data for initial load into the database.  */

    /* Define descriptors and prepare records for */
    /* loading into the database.                 */
    if  clusters to be formed at file level
    then
        perform  FILE_PREP
    else
        perform DATABASE_PREP
    end if;

    /* Sort data into clusters.       */
    perform  SORT_INTO_CLUSTERS;

    /* Load data into database store. */
    perform  LOAD_DATA;
 end subsystem;
```

Second Level Specifications for Database Load

```
4.1  proc  FILE_PREP;      /* FILEPREP (DBL11) */

     /* Prepare files for loading to the database store. Clusters will be */
     /* defined at file level.                                             */

4.2      array  type-C_attr_names;   /* Attribute names over which type-C */
                                     /* descriptors will be defined.      */
4.3      arglist = (clustering_level, /* "FILE"                            */
                    record_type,      /* Type records in the current file. */
                                      /* (Payroll, Employee, Inventory ...)*/
                    database_name,    /* Generic name for this database,   */
                                      /* e.g. PERSONNEL, PARTS, CONTRACTS. */
                    pointer to type-C_attr_names);

4.4      scalar  atpointer              /* Pointer to instance of AT        */
                                        /* created for this task.           */
                 rectemppointer;        /* Pointer to RTEMP for current     */
                                        /* file.                            */


4.5      arglist.clustering_level := "FILE";
4.6      get arglist.database_name from terminal;

4.7      while  more files to be loaded do
4.8      begin

4.9          get arglist.record_type from terminal;

             /* Define all descriptors for this file.          */
             /* Argument list constructed as above is passed to */
             /* the DEFINE_DESCRIPTORS procedure, which returns */
             /* a pointer to the instance of AT created for this*/
             /* task.                                           */

4.10         perform  DEFINE_DESCRIPTORS(arglist,
                                         atpointer);

4.11         arglist + atpointer;

             /* Define record structure via a record template.  */
             /* Argument list constructed as above is passed to */
             /* this procedure, which returns a pointer to the  */
             /* record template created for this file.          */

4.12         perform  DEFINE_RECTEMP(arglist,
                                     rectemppointer);

4.13         arglist + rectemppointer;

             /* Examine each record in the file, and determine     */
             /* the set of descriptor ids representing descriptors */
             /* from which keywords in that record can be derived. */
             /* Create records to be sorted into clusters.         */
             /* Argument list constructed as above is passed to    */
             /* this procedure.                                    */

4.14         perform  DERIVE_DIRECTORY_KEYWORDS(arglist);

4.15     end while

4.16 end proc;
```

```
6.1  proc  DATABASE_PREP;        /* DBPREP (DBL12) */

     /* Prepare data for loading to the database store.  Clusters will be
        defined at database level.   */

6.2       array  type-C_attr_names;   /* Attribute names over which type-C */
                                      /*  descriptors will be defined.     */
6.3       arglist = (clustering_level, /* "DATABASE"                       */
                     record_type,      /* Type records in the current file. */
                                       /* (Payroll, employee, inventory ...)*/
                     database_name,    /* Generic name for this database,   */
                                       /* e.g. PERSONNEL, PARTS, CONTRACTS. */
                     pointer to type-C_attr_names);

6.4       scalar  atpointer,              /* Pointer to instance of AT        */
                                          /* created for this task.           */
                  rectemppointer;         /* Pointer to RTEMP for current     */
                                          /* file.                            */


6.5       arglist.clustering_level := "DATABASE";

6.6       get arglist.database_name from terminal;

          /* Define all descriptors for this database.       */
          /* Argument list constructed as above is passed to */
          /* the DEFINE_DESCRIPTORS procedure, which returns */
          /* a pointer to the instance of AT created for this*/
          /* task.                                           */

6.7       perform  DEFINE_DESCRIPTORS(arglist,
                                      atpointer);

6.8       arglist + atpointer;

6.9       while  more files in this database do
6.10      begin

6.11          get arglist.record_type from terminal;

              /* Define record structure via a record template.  */
              /* Argument list constructed as above is passed to */
              /* this procedure, which returns a pointer to the  */
              /* record template created for this file.          */

6.12          perform  DEFINE_RECTEMP(arglist,
                                      rectemppointer);

              /* Examine each record in the file, and determine     */
              /* the set of descriptor ids representing descriptors */
              /* from which keywords in that record can be derived. */
              /* Create records to be sorted into clusters.         */
              /* Argument list constructed as above is passed to    */
              /* this procedure.                                    */

6.13      arglist + rectemppointer;

6.14      perform  DERIVE_DIRECTORY_KEYWORDS
                           (arglist);

6.15      end while

6.16 end proc;
```

```
8.1  proc  SORT_INTO_CLUSTERS;      /* SRTCLUST  (DBL13) */

     /* Records to be sorted have the form:                            */
     /*  record = (descr_count,      Number of descriptors in descr_ids*/
     /*                              list which follows.               */
     /*             descr_ids,       List of descriptor ids from which */
     /*                              this record can be derived.       */
     /*             database_record);  Record formatted into form      */
     /*                              required for storing in the data- */
     /*                              base store.                       */

8.2      open files;

8.3      sort records in ascending sequence by
                   descriptors_ids within descr_count;

8.4      close files;

8.5  end proc;


9.1  proc  LOAD_DATA;               /* LOADDATA  (DBL14) */

     /* Distribute clusters of data across the multiple backends   */
     /* according to the track-splitting-with-random-placement     */
     /* data placement strategy.                                   */

9.2      scalar  cdtpointer;            /* From CDTM module CREATE. */

9.3      record = (descr_count,      /* Number of descriptors in descr_ids*/
                                     /* list which follows.               */
                   descr_ids,        /* List of descriptor ids from which */
                                     /* this record can be derived.       */
                   database_record); /* Record already formatted into form*/
                                     /* required for storing in the data- */
                                     /* base store.                       */

9.4      system_info = (number_of_backends,   /* This data assumed to be */
**                      backend_addresses,    /*available in some system*/
                        track_capacity);      /* generation file accessi-*/
                                              /* ble through some module */
                                              /* called SYSDATA.         */

         /* Create an instance of CDT for this task.  The CREATE function of*/
         /* the CDT module returns a pointer to the instance of CDT created */
         /* for this database. */

9.5      perform  CDTM$CREATE(cdtpointer);

         /* Get from the system the information required for this task. */

9.6      perform  SYSDATA$INFO(system_info);

         /* Read the first record.  */

9.7      open file of sorted records and read first record;


9.8      while  more clusters in sort file do
9.9      begin

9.10         perform PROCESS_A_CLUSTER(record,
                                       system_info,
                                       cdtpointer);
9.11     end while

9.12 end proc;
```

Third Level Specifications for Database Load

```
4.10.1  proc  DEFINE_DESCRIPTORS   /* DESCRDEF (DBL111) */
              (input: inpointer(clustering_level,
                                record_type,
                                database_name,
                                type-C_attr_names),
               output: atpointer);

        /* Define descriptors for this file or database, and store them  */
        /* in the DDIT created for this task.                             */
        /*  Input is a pointer to an list which contains                  */
        /*     clustering_level ("FILE" or "DATABASE"), record_type,      */
        /*     database_name, a list of attribute names over which type-C*/
        /*     descriptors are to be defined.                             */
        /*  Output is a pointer to the instance of AT created for         */
        /*  this task.                                                    */

4.10.2      scalar name,            /* Name to identify AT              */
                   atpointer;       /* Pointer to instance of AT        */
                                    /* created for this task.           */

4.10.3      if inpointer.clustering_level = "database"
4.10.4      then
4.10.5          name := inpointer.database_name;   /* NOTE:  may also want */
4.10.6      else                                   /* to indicate cluster- */
4.10.7          name := inpointer.record_type;     /* ing level here.      */
4.10.8      end if;

            /* CREATE function of module ATM returns a pointer to */
            /* the instance of AT created for this task.          */

4.10.9      perform ATM$CREATE(name, atpointer);

4.10.10     if  unsuccessful create
4.10.11     then
4.10.12         display message
4.10.13     else

4.10.14         begin

                /* First all type-A descriptors.  */
4.10.15         while  more type-A descriptors do
4.10.16            perform  DEF_TYPE-A_DESCR(atpointer);
4.10.17         end while;

                /* Then all type-B descriptors.  */
4.10.18         while  more type-B descriptors do
4.10.19             perform  DEF_TYPE-B_DESCR(atpointer);
4.10.20         end while;

               /* Build an array of attribute names over which type-C    */
               /* descriptors are to be defined when input is read.      */

4.10.21         perform  LIST_TYPE-C_ATTR_NAMES
                                   (inpointer.type-C_attr_names,
                                    atpointer);

               /* Allow user to review descriptors for accuracy.         */

4.10.22         perform   REVIEW_DESCRIPTORS
                                          (inpointer.type-C_attr_names,
                                           atpointer);

4.10.23         return(atpointer);

4.10.24     end if;

4.10.25 end proc;
```

```
4.12.1  proc DEFINE_RECTEMP        /* RTEMPDEF (DBL112) */
             (input: inpointer(clustering_level,
                               record_type,
                               database_name,
                               type-C_attr_names,
                               atpointer),
              output: rectemppointer);

        /* Define the structure of the records in this file by          */
        /* building a record template.                                  */

        /*  Input is a pointer to an list which contains                */
        /*     clustering_level ("FILE" or "DATABASE"), record_type,    */
        /*     database_name, a list of attribute names over which type-C*/
        /*     descriptors are to be defined, and a pointer to the   AT. */
        /*  Output is a pointer to the record template created          */
        /*  for this task.                                              */

4.12.2       scalar record_type,         /* Type of records in this file.  */
                    attr_name,           /* Attribute name.                */
                    descr_type,          /* A, B, C, or NOTFOUND.          */
                    rectemppointer,      /* Pointer to RTEMP created for   */
                                         /* this task.                     */
                    dditpointer,         /* Pointer into the DDIT returned */
                                         /* from the FIND function of AT.  */
                    duplicate,           /* Indicator - TRUE or FALSE.     */
                    matched,             /* Indicator - TRUE or FALSE.     */
                    successful;          /* Indicator - TRUE or FALSE.     */

            /* Attrlist to be returned from GET_ATTRIBUTE_CHARACTERISTICS */
            /* procedure has the form:                                     */
4.12.3       attrlist = (attr_name, attr_data_type, attr_format,       */
                        attribute_characteristics);


4.12.4       record_type := inpointer.record_type;

             /* Invoke the CREATE function of module RTEMPM to  */
             /* create an record template for this task.  A     */
             /* pointer to the template is returned.            */

4.12.5       perform   RTEMPM$CREATE(record_type, rectemppointer);

4.12.6       if  create is not successful
4.12.7       then
4.12.8          display message;
4.12.9       else
4.12.10         begin
4.12.11         while  more attributes to be defined do
4.12.12         begin
4.12.13            get attr_name from terminal;

                   /* Check to see whether this attribute name is */
                   /* unique within this record.                  */

4.12.14            perform  RTEMPM$DUPCHECK(rectemppointer,
                                            attr_name,
                                            duplicate);

                   /* If it is unique, then get the characteristics*/
                   /* of the attribute.                            */

4.12.15            if  duplicate is TRUE
4.12.16            then
4.12.17               display message
4.12.18            else
4.12.19               begin

                      /* Get attribute characteristics, and return a  */
                      /* list of attribute characteristics, attrlist. */
```

```
4.12.20                 perform  GET_ATTRIBUTE_CHARACTERISTICS
                                                   (attr_name,
                                                    attrlist);
                       /* Determine whether there is a descriptor defined */
                       /* over this attribute name.  If there is, mark the*/
                       /* record template entry to indicate type.  This   */
                       /* mark will be used in DERIVE_DIRECTORY_KEYWORDS */
                       /* procedure.                                      */

4.12.21                 perform  ATM$FIND(attr_name,
                                              descr_type,
                                              dditpointer);

4.12.22                 if  a descriptor has been defined
                              for this attribute name
                                  /* descr_type not = NOTFOUND */
4.12.23                 then
4.12.24**                   attrlist + descr_type;
4.12.25                 else
4.12.26                     begin
4.12.27                     perform  SEARCH_TYPE-C_ATTR_NAME
                                       (inpointer.type-C_attr_names,
                                        attr_name,
                                        matched);
4.12.28                     if  matched is TRUE
4.12.29                     then
4.12.30**                       attrlist + 'C';
4.12.31                     else
4.12.32                         attrlist + nullcharacter;
4.12.33                     end begin;  /* Not type A or B. */
4.12.34                 end if;         /* If descriptor defined. */

                    /* Add information about this attribute to the  */
                    /* template.                                     */

4.12.35                 perform  RTEMPM$INSERT(rectemppointer,
                                                   attrlist,
                                                   successful);

4.12.36                 if successful is FALSE
4.12.37                 then
4.12.38                     display message;
4.12.39                 end if;

4.12.40         end while;  /* While more attributes to be defined. */
4.12.41        end begin;

           /* Allow the user to review the entire template. */

4.12.42     perform  REVIEW_RECTEMP(rectemppointer);

4.12.44     end if;

4.12.45 end proc;


4.14.1  proc DERIVE_DIRECTORY_KEYWORDS       /* DRVKWORD (DBL113) */
             (input: inpointer(clustering_level,
                               record_type,
                               database_name,
                               type-C_attr_names,
                               atpointer,
                               rectemppointer));


        /* Input is a pointer to a list of arguments which contains */
        /* clustering level ("FILE" or "DATABASE"), record type,    */
        /* database name, type-C_attr_names, (a list of attribute   */
        /* names over which type-C descriptors are to be defined), a*/
        /* pointer to the AT and a pointer to the RTEMP for this    */
        /* file.                                                    */
```

```
          /* The procedure reads records from the source file(s).     */
          /* For each record, the set of descriptors from which the   */
          /* record is derivable is determined, the record is format- */
          /* ted into the form required for storage in the database,  */
          /* and a count of descriptors, the descriptor list, and     */
          /* the formatted record are written to a file for sorting.  */

4.14.2        scalar  descr_id,      /* Descriptor id.                 */
                      descr_count,   /*   of descriptors derived.      */
                      field_count,   /* Indicates nth field of record.*/
                      field_value,   /* Value of nth field.            */
                      filename;      /* Name of current input file.    */

4.14.3        array  descriptor_ids;   /* List of descriptors derived.*/

4.14.4        predicate = (attribute, "=", value);  /* Equality predicate */
                                      /* to test derivation of keyword.*/

4.14.5        input_record = (field_value$field_value$ ... field_value );

4.14.6        rectemp_entry = (attr_name, attr_data_type, lower_bound,
                        upper_bound, descr_ind);    /* Entry re-   */
                                      /* trieved from RTEMP.        */


4.14.7        open output file for records to be sorted

4.14.8        while  more files of input data do
4.14.9        begin
4.14.10           get filename from terminal;
4.14.11           open file filename;

4.14.12           while  more records in file do
4.14.13           begin
                      /* Initialize counts, get first record.   */
4.14.14               descr_count := 0;
4.14.15               field_count := 0;
4.14.16               get input_record from file filename;

4.14.17               while  more fields in record do
4.14.18               begin

                          /* Get a field value from the record.  */

4.14.19                   field_value := next field_value from input_record;
4.14.20                   field_count := field_count + 1;

                          /* Get the entry from the record template */
                          /* which contains attribute name and char-*/
                          /* acteristics of that field.             */

4.14.21                   perform  RTEMPM$GETENTRY(inpointer.rectemppointer,
                                                   field_count,
                                                   rectemp_entry);

4.14.22                   if rectemp_entry.descr_ind not = null
4.14.23                   then
4.14.24                       begin

                              /* Build a keyword predicate to be used to test */
                              /* whether current attribute-value pair can be  */
                              /* derived from any descriptor.                 */

4.14.25                       predicate.attribute := rectemp_entry.attr_name;
4.14.26                       predicate.value := field_value;

                              /* Determine whether keyword is derivable.     */
                              /* Descriptor id will be updated by the        */
                              /* DERIVE procedures called below.  If the     */
                              /* descriptor id is null, the keyword is not   */
                              /* derivable.                                  */
```

```
4.14.27                     if rectemp_entry.descr_ind = 'x'
**                          /* 'x' indicates that descriptor is type A or B.*/
4.14.28                     then
4.14.29                         perform  DERIVE_FROM_A_OR_B_DESCR
                                           (predicate,
                                            inpointer.atpointer,
                                            descr_id);

4.14.30                     else
4.14.31                         perform  DERIVE_FROM_C_DESCR
                                           (predicate,
                                            inpointer.atpointer,
                                            descr_id);
4.14.32                     end if;

                            /* If it is derivable, insert the corresponding */
                            /* descriptor id into the list of such ids being*/
                            /* built for this record.                       */

4.14.33                     if  keyword predicate is derivable
4.14.34                     then
4.14.35                         begin
4.14.36                         perform  PUT_DESCR_ID_INTO_LIST
                                                             (descr_id,
                                                              descriptor_ids
                                                              descr_count);
4.14.37                         descr_count := descr_count + 1;
4.14.38                      end if;

4.14.39                  end if;

4.14.40              end while;   /* more fields loop */

4.14.41              perform  BUILD_SORT_RECORD(descr_count,
                                              descriptor_ids,
                                              input_record,
                                              rectemppointer);
4.14.42          end while;  /* records in file loop */

4.14.43          close file filename;

4.14.44      end while;     /* files to be sorted loop */

             /* Review the list of type-C attribute names, and create  */
             /* null AT entries where no descriptors have been defined.*/

4.14.45      perform  REVIEW_TYPE-C_ATTR_NAMES
                                      (type-C_attr_names,
                                       atpointer);

4.14.46      close file for sort records;

4.14.47  end proc;


9.10.1    proc PROCESS_A_CLUSTER            /* PROCLUST (DBL141) */
**                      (input: record,
                                system_info,
                                cdtpointer);

          /* Process a cluster for loading into the database store.   */
**        /* Input is the first record of a cluster, some system      */
**        /* information, and a pointer to the CDT created for this   */
**        /* task.                                                    */
          /* Records have the form:                                   */
          /*   record = (descr_count, descriptor_ids, database_record)*/
```

```
**        /* System information is a list of backend addresses.         */
**        /*   system_info = (number_of_backends,                        */
**        /*                  backend_addresses,                         */
          /*                  track_capacity);                           */


9.10.1    scalar  next_backend_index,    /* Index into array of back-*/
                                         /* end addresses in system_info*/
                  capacity_remaining,    /* Capacity remaining in the*/
                                         /* track of the next backend*/
                                         /* at which records of the  */
                                         /* current cluster are to   */
                                         /* be stored. Used to update*/
                                         /* CINBT.                   */
                  cluster_number;


         /* Randomly select a backend at which to begin distribution of */
         /* records in the first cluster, and set the track capacity.   */

9.10.2    perform   GET_RANDOM_BACKEND_START
**                     (system_info.number_of_backends,
                        next_backend_index);

9.10.3**  capacity := system_info.track_capacity;

          /* Generate a Cluster Definition Table entry for the new cluster. */
          /* The following procedure returns the cluster number.            */

9.10.4     /* From somewhere as yet undefined, get a new cluster number.    */

9.10.5     perform   CDTM$INSERTNEWCLUSTER
**                                   (record.descriptor_ids,
                                      cdtpointer,
                                      cluster_number);


          /* Physically distribute the data over the multiple    */
          /* backends according to the selected data placement   */
          /* strategy. The DISTRIBUTE_RECORDS procedure, starting*/
          /* at a randomly selected backend, evenly distributes  */
          /* data across the backends in track-size lots.  The   */
          /* address of the next backend and the amount of stor- */
          /* age available there are returned from the procedure */
**        /* to be used to update the CINBT.  Note that the      */
**        /* records are read ahead, so that upon return from    */
**        /* the DISTRIBUTE_RECORDS procedure, the first record  */
**        /* of a new cluster will have replaced the record ori- */
**        /* ginally passed.                                     */

9.10.6**   perform   DISTRIBUTE_RECORDS(record,
**                                  next_backend_index,
**                                  system_info,
**                                  cluster_number,
                                    capacity_remaining);


         /* Update the Cluster-ID-to-Next-Backend-Table with the address*/
         /* of the next backend into which records belonging to this    */
         /* cluster should be inserted, and the remaining capacity at   */
         /* that backend.                                               */

9.10.7    perform  CINBTM$UPDATE(cluster_number,
                         next_backend_index,
                         capacity_remaining);

9.10.8  end proc;
```

Fourth Level Specifications for Database Load

```
4.10.16.1   proc  DEF_TYPEA_DESCR            /* TYPEADEF (DBL1111)  */
                             (input: atpointer);

            /* Define all type A descriptors.  Input is a pointer to */
            /* the instance of AT created for this task.             */


4.10.16.2   scalar  attr_data_type, /* Character, integer, etc.      */
                    descr_id,       /* Descriptor id.                */
                    descr_type,     /* A, B, C, or NOTFOUND.         */
                    attr_name,      /* Attribute name.               */
                    duplicate,      /* Indicator, TRUE or FALSE.     */
                    dditpointer;    /* Pointer into the DDIT, either */
                                    /* to first descriptor defined   */
                                    /* for this attribute or to last */
                                    /* descriptor inserted.          */

4.10.16.3   descriptor = (lower_bound,  upper_bound);
            /* An "other" descriptor will be defined for each attribute   */
            /* over which descriptors are defined, to represent all those */
            /* keywords which are not deriveable from any other descriptor*/
            /* defined for that attribute.                                */
4.10.16.4   other_descriptor = (lower_bound, upper_bound);


              /* Initialize "other" descriptor bounds. */
4.10.16.5     other_descriptor.lower_bound = null;
4.10.16.6     other_descriptor.upper_bound = null;

4.10.16.7     get attr_name from terminal;
4.10.16.8     get attr_data_type from terminal;
4.10.16.9     while  more descriptors for this attribute do
4.10.16.10    begin
                /* NOTE:  Limits supplied for the descriptor must */
                /* be right-justified, padded on left.            */
4.10.16.11      get descriptor.lower_bound from terminal;
4.10.16.12      get descriptor.upper_bound from terminal;

4.10.16.13      check upper and lower bounds to insure that data is
                    of the correct type;

4.10.16.14      if  data not of correct type
4.10.16.15      then
                    display an informational message;
4.10.16.17      else

4.10.16.18          begin
4.10.16.19**        duplicate := false;
4.10.16.20          descr_type := null;
4.10.16.21          perform ATM$FIND(attr_name,
                                     dditpointer,
                                     pointer to descr_type);

4.10.16.22          if this attribute name found in AT
                         /* descr_type not = NOTFOUND */
4.10.16.23          then
4.10.16.24              begin
4.10.16.25              perform DDITM$DUPCHECK
                                           (descriptor,
                                            dditpointer,
                                            duplicate);

4.10.16.26              if  this descriptor exactly duplicates
                            another or the range overlaps another
                            /* duplicate is TRUE */
4.10.16.27              then
4.10.16.28                  display message;
4.10.16.29              else
                            /* Get descr_id from somewhere as yet */
                            /* undefined.                         */
```

```
4.10.16.30                      perform DDITM$INSERT(descriptor,
                                                     dditpointer,
                                                     descr_id);
4.10.16.31                  end if
4.10.16.32                  end begin
4.10.16.33             else
4.10.16.34                 begin
                           /* Insert type "other" descriptor for each new  */
                           /* attribute name added to the AT.              */
                           /* Get descr_id from somewhere as yet undefined.*/
4.10.16.35                 perform DDITM$INSERT(other_descriptor,
                                                dditpointer,
                                                descr_id);
                           /* Now insert the new attribute name.          */
4.10.16.36                 perform ATM$INSERT(attr_name,
                                                 'A',
                                                  dditpointer);
                           /* Now insert the new descriptor defined here.  */
                           /* Get descr_id from somewhere as yet undefined.*/
4.10.16.37                 perform DDITM$INSERT(descriptor,
                                                dditpointer,
                                                descr_id);
4.10.16.38                 end begin;
4.10.16.39             end if;   /* (in AT) */
4.10.16.40             end begin;
4.10.16.41         end if;  /* (not of correct type) */

4.10.16.42     end while;

4.10.16.43 end proc;


4.10.19.1  proc  DEF_TYPEB_DESCR          /* TYPEBDEF  (DBL1112)  */
                               (input: atpointer);

           /* Define all type B descriptors.  Input is a pointer to */
           /* the instance of AT created for this task.             */

4.10.19.2  scalar  attr_data_type, /* Character, integer, etc.      */
                   descr_id,       /* Descriptor id.                */
                   descr_type,     /* A, B, C, or NOTFOUND.         */
                   attr_name,      /* Attribute name.               */
                   duplicate,      /* Indicator, TRUE or FALSE.     */
                   dditpointer;    /* Pointer into the DDIT, either */
                                   /* to first descriptor defined   */
                                   /* for this attribute or to last */

4.10.19.3  descriptor = (lower_bound, upper_bound);
           /* An "other" descriptor will be defined for each attribute   */
           /* over which descriptors are defined, to represent all those */
           /* keywords which are not deriveable from any other descriptor*/
           /* defined for that attribute.                                */

4.10.19.4  other_descriptor = (lower_bound, upper_bound);


               /* Initialize "other" descriptor bounds. */
4.10.19.5      other_descriptor.lower_bound = null;
4.10.19.6      other_descriptor.upper_bound = null;

4.10.19.7      get attr_name from terminal;
4.10.19.8      get attr_data_type from terminal;
               /* NOTE:  Limits supplied for the descriptor must */
               /* be right-justified, padded on left.            */
4.10.19.9      while  more descriptors for this attribute do
4.10.19.10     begin
                   descriptor.lower_bound := null;
4.10.19.11         descriptor.lower_bound := null;
4.10.19.12         get descriptor.upper_bound from terminal;

4.10.19.13         check upper bound to insure that data is
                       of the correct type;
```

```
4.10.19.14          if  data not of correct type
4.10.19.15          then
4.10.19.16              display local message;
4.10.19.17          else
4.10.19.18              begin
4.10.19.19*             duplicate = false;
4.10.19.20              descr_type := [illegible]
4.10.19.21              perform ATM$FIND(attr_name,
                                         dditpointer,
                                         pointer to descr_type);

4.10.19.22              if this attribute name found in AT
4.10.19.23              then
4.10.19.24                  begin
4.10.19.25                  perform DDITM$DUPCHECK(descriptor,
                                                   dditpointer,
                                                   duplicate);
4.10.19.26                  if  this descriptor exactly duplicates
                                another or the range overlaps another
4.10.19.27                  then
4.10.19.28                      display message;
4.10.19.29                  else
                                /* Get descr_id from somewhere as yet */
                                /* undefined.                         */
4.10.19.30                      perform DDITM$INSERT(descriptor,
                                                     dditpointer,
                                                     descr_id);
4.10.19.31                  end if;
4.10.19.32                  end begin;
4.10.19.33              else
4.10.19.34                  begin
                            /* Insert type "other" descriptor for each new  */
                            /* attribute name added to the AT.              */
                            /* Get descr_id from somewhere as yet undefined.*/
4.10.19.35                  perform DDITM$INSERT(other_descriptor,
                                                 dditpointer,
                                                 descr_id);
                            /* Now insert the new attribute name.  */
4.10.19.36                  perform ATM$INSERT(attr_name,
                                               'B',
                                               dditpointer);
                            /* Now insert the new descriptor defined here.  */
                            /* Get descr_id from somewhere as yet undefined.*/
4.10.19.37                  perform DDITM$INSERT(other_descriptor,
                                                 dditpointer,
                                                 descr_id);
4.10.19.38                  end begin;

4.10.19.39              end if       /* (in AT) */
4.10.19.40              end begin

4.10.19.41          end if;  /* (not of correct type) */

4.10.19.42      end while;

4.10.19.43 end proc;


4.10.21.1  proc  LIST_TYPE-C_ATTR_NAMES       /* TYPECLST  (DBL1113) */
                                (input: type-C_attr_names,
                                        atpointer);
           /* List all the attribute names over which type-C descriptors */
           /* are to be defined.  Input is a list for  attribute names   */
           /* over which type-C attributes are to be defined, and a      */
           /* pointer to the AT.                                         */

4.10.21.2       scalar  index,       /* Index to list of attribute names.  */
                        attr_name,
**                      duplicate,   /* Indicator - TRUE or FALSE.         */
                        dditpointer,/* Pointer into DDIT returned from ATM*/
                                     /* FIND function.                     */
                        descr_type; /* A, B, C, or NOTFOUND.              */
```

```
4.10.21.3       index := 1;         /* Null indicates end of list.  */
4.10.21.4       type-C_attr_names[index] := null;

4.10.21.5       while  more type-C descriptors do
4.10.21.6       begin
4.10.21.7           get attr_name from terminal;
4.10.21.8           perform  ATM$FIND(attr_name,
                                      dditpointer,
                                      pointer to descr_type);
4.10.21.9           if a type-A or type-B descriptor is already defined
                        over this attribute name
                        /* descr_type not = NOTFOUND  */
4.10.21.10          then
4.10.21.11             display error message;
4.10.21.12          else
4.10.21.13             begin
4.10.21.14**           duplicate = FALSE;
4.10.21.15             perform  SEARCH_TYPE-C_ATTR_NAMES
                                 (type-C_attr_names,
                                  attr_name,
                                  duplicate);
4.10.21.16             if  duplicate is FALSE
4.10.21.17             then
4.10.21.18                 begin
4.10.21.19                 type-C_attr_names[index] := attr_name;
4.10.21.20                 index := index + 1;
4.10.21.21                 type-C_attr_names[index] := null;
4.10.21.22             end if;
4.10.21.23          end if;

4.10.21.24      end while;

4.10.21.25 end proc;


4.10.22.1  proc  REVIEW_DESCRIPTORS        /*  REVDESCR  (DBL1114) */
                         (input: type-C_attr_names,
                                 atpointer);

           /* S T U B  */

4.10.22.?  end proc;


4.12.19.1  proc  GET_ATTRIBUTE_CHARACTERISTICS /* ATTRCHAR  (DBL1121) */
                        (input: attr_name, attrlist);

           /* Get characteristics of an attribute for and entry */
           /* in the record template.                           */
           /* Input to the procedure is an attribute name and a */
           /* list for attribute characteristics.               */
           /* The values of those characteristics will be col-  */
           /* lected in this procedure.                         */
           /* Attribute list has the form:                      */
           /*  attrlist = (attr_name,        Attribute name.          */
           /*             value_data_type,  String   , integer, float ..*/
           /*             value_format,     Fixed or variable (string)  */
           /*             value_char1,      First characteristic.       */
           /*             value_char2);     Second characteristic.      */


4.12.19.2        attrlist.attr_name := attr_name;
4.12.19.3        get attrlist.value_data_type from terminal;

4.12.19.4        case attrlist.value_data_type value

4.12.19.5           integer:
4.12.19.6              begin
4.12.19.7              attrlist.value_format := null;
```

```
4.12.19.8               get attrlist.value_char1;  /* Min value.  */
4.12.19.9               get attrlist.value_char2;  /* Max value.  */
4.12.19.10              end begin;

4.12.19.11           string:
4.12.19.12              begin
                        /* Fixed or variable length string ?          */
4.12.19.13              get attrlist.value_format from terminal;
4.12.19.14              if attrlist.value_format is fixed
4.12.19.15              then
4.12.19.16                 begin
                           /* Min length = 0; get max length.         */
4.12.19.17                 attrlist.value_char1 := 0;
4.12.19.18                 get attrlist.value_char2 from terminal;
4.12.19.19                 end begin
4.12.19.20              else
4.12.19.21                 begin
                          /* Get min and max lengths.                 */
4.12.19.22                get attrlist.value_char1 from terminal;
4.12.19.23                get attrlist.value_char2 from terminal;
4.12.19.24              end if;
4.12.19.25              end begin;

4.12.19.26           float: ;

4.12.19.27           otherwise: ;

4.12.19.28        end case

4.12.19.29 end proc;


4.12.27.1  proc  SEARCH_TYPE-C_ATTR_NAMES      /* SRCHCLST  (DBL1122) */
                        (input: type-C_attr_names,
                                attr_name,
**                       output: found);
           /* Search the list of attribute names over which type C  */
           /* descriptors are to be defined to determine whether    */
           /* attr_name is a duplicate.  Input is a list of attri  */
           /* bute names over which type-C descriptors are to be    */
           /* defined, and an attribute name.                       */

4.12.27.2       scalar  index,   /* Index into list of attribute names.*/
**                      found;   /* Indicator, TRUE or FALSE.          */

4.12.27.3       index := 1;
4.12.27.4**     found := false;

4.12.27.5       while  type-C_attr_names[index] not = null
                          /* null indicates end of list */
                       and
                       type-C_attr_names[index] not = attr_name do
4.12.27.6
4.12.27.7           index := index + 1;
4.12.27.8       end while;

4.12.27.9       if  type-C_attr_names[index] = attr_name
4.12.27.10      then
4.12.27.11**        found := true;
4.12.27.12      end if;

4.12.27.13  end proc;


4.12.42.1  proc  REVIEW_RECTEMP          /*  REVRTEMP  (DBL1123) */
                          (input: rectemppointer);

           /* STUB */

4.12.42.? end proc;
```

```
4.14.29.1  proc  DERIVE_FROM_A_OR_B_DESCR   /* DRVAORB  (DBL1131) */
                   (input: predicate, atpointer,
                    output: descr_id);

           /* Determine whether there exists a type A or type B descriptor */
           /* from which the current keyword can be derived.               */
           /* Input is an equality predicate, and a pointer to the AT.     */
           /* A predicate has the form:                                    */
           /*  predicate = (attribute, "=", value)                         */
           /* A descriptor id is returned to the calling procedure.        */

4.14.29.2      scalar  descr_id,
                       dditpointer, /* Pointer into DDIT returned from    */
                                    /* FIND function of ATM.              */
                       descr_type;     /* A, B, C, or NOTFOUND.          */


4.14.29.3      descr_id := null;

               /* FIND returns a pointer to first descriptor defined for  */
               /* this attribute name. */

4.14.29.3      perform ATM$FIND(predicate.attribute,
                                dditpointer,
                                pointer to descr_type);

               /* DERIVE returns the descriptor id for any descriptor from */
               /* which this keyword can be derived. (May be null)         */


4.14.29.4      perform DDITM$DERIVE(predicate,
                                dditpointer,
                                descr_id);


4.14.29.5   end proc;


4.14.31.1  proc  DERIVE_FROM_C_DESCR          /* DRVC  (DBL1132) */
                    (input: predicate, atpointer,
                     output: descr_id)
           /* Determine whether keyword can be derived from an existing    */
           /* type-C descriptor. If not, define a new type-C descriptor.  */
           /* In put is an equality predicate, and a pointer to the AT.    */
           /* A predicate has the form:                                    */
           /*  predicate = (attribute, "=", value)                         */
           /* A descriptor id is returned to the calling procedure.        */


4.14.31.2      scalar  descr_id,    /* Descriptor id returned from DERIVE  */
                                    /* function of DDITM.                 */
                       dditpointer, /* Pointer into DDIT returned from    */
                                    /* FIND function of ATM.              */
                       keep_ddit_ptr,  /* Save pointer returned fromFIND  */
                                    /* to compare with that returned by   */
                                    /* INSERT function of DDITM.          */
                       descr_type;  /* A, B, C, or NOTFOUND.              */

4.14.31.3      descriptor = (lower_bound, upper_bound);


4.14.31.4      dditpointer := null;

               /* FIND returns pointer to the first descriptor defined for */
               /* this attribute name in DDIT.                              */

4.14.31.5      perform  ATM$FIND(predicate.attribute,
                                 dditpointer,
                                 pointer to descr_type);
```

```
4.14.31.6      descr_id := NULL;

               /* If this is the first type-C descriptor defined for this */
               /* name, first insert the descriptor into DDIT, then pu    */
               /* an entry in the AT with a pointer to that descriptor.   */
               /* If this is not the first type-C descriptor defined for  */
               /* this attribute name, check to see whether this descrip- */
               /* tor already exists.  If it does, use the existing id;   */
               /* otherwise, insert the new descriptor into DDIT.         */

4.14.31.7      if  no descriptors yet defined for this attribute
4.14.31.8      then
4.14.31.9          begin
4.14.31.10         descriptor.lower_bound = null;
4.14.31.11         descriptor.upper_bound = predicate.value;
                   /* Get descr_id from somewhere as yet undefined.  */
4.14.31.12         perform DDITM$INSERT(descriptor,
**                                      dditpointer,
                                        descr_id);
4.14.31.13         perform ATM$INSERT(predicate.attribute,
                                      'C',
                                 dditpointer);
4.14.31.14         end begin
4.14.31.15     else     /* Descriptors previously defined for this attr.*/
4.14.31.16         begin
4.14.31.17         keep_ddit_ptr := dditpointer;
                   /* Does this descriptor already exist?  */
4.14.31.18         perform DDITM$DERIVE(predicate
                                        dditpointer,
                                        descr_id);
                   /* If not, add it.                      */
4.14.31.19         if keyword is not derivable
                   /* This is a new descriptor             */
4.14.31.20         then
4.14.31.21             begin
                       /* Get descr_id from somewhere as yet undefined. */
4.14.31.22             perform DDITM$INSERT(descriptor,
**                                          dditpointer,
                                            descr_id);
4.14.31.23             if keep_ddit_ptr != dditpointer
4.14.31.24             then
4.14.31.25**               perform ATM$UPDATE(predicate.attribute,
                                              dditpointer);
4.14.31.26             end begin
4.14.31.27         end if;
4.14.31.28         end begin;
4.14.31.29     end if;

4.14.31.30 end proc;


4.14.36.1  proc  PUT_DESCR_ID_INTO_LIST       /* PUTINLST  DBL1133) */
                      (input: descr_id, descriptor_ids, descr_count);

           /* Insert a new descriptor id into the list of descriptor ids */
           /* from which the current record can be derived.              */
           /* Input is a descriptor id, a list of descriptor ids, and a  */
           /* count of the number of items in the list.  The list must be*/
           /* a new id to be inserted into the list.  The list must be   */
           /* maintained in ascending sequence.                          */

4.14.36.2      insert descr_id in order into list of descriptor ids

4.14.36.3  end proc;
```

```
4.14.41.1  proc  BUILD_SORT_RECORD            /* BLDSRT  (DBL1134) */
                        (input: descr_count,
                                descriptor_ids,
                                input_record,
                                rectemppointer);

4.14.41.2      sortrec = (descr_count, descriptor_ids, database_record);


4.14.41.3      sortrec.descr_count := descr_count;
4.14.41.4      sortrec.descriptor_ids := descriptor_ids;

4.14.41.5      format input_record into sortrec.database_record;

4.14.41.6      write sortrec to output file for sort

4.14.41.7  end proc;


4.14.45.1  proc  REVIEW_TYPE-C_ATTR_NAMES     /* REVTYPEC  (DBL1134) */
                          (input: type-C_attr_names,
                                  atpointer);

           /* Review the list of attribute names over which type-C        */
           /* descriptors are to be defined.  If no descriptors have yet  */
           /* been defined for an attribute, create an entry in AT with   */
           /* a null pointer in place of a pointer into DDIT.             */
           /* Input is a list of the attribute names over which type-C    */
           /* descriptors are to be defined and a pointer to the AT.      */

4.14.45.2      scalar  index,          /* Index into list of attribute names.*/
                       dditpointer,    /* Pointer from AT into DDIT.         */
                       descr_type;     /* A, B, C, or NOTFOUND.             */


4.14.45.3      index := 1;

4.14.45.4      while  type-C_attrnames[index] not = null do
                   /* null indicates end of list */
4.14.45.5        begin
4.14.45.6        perform ATM$FIND(type-C_attr_names[index],
                                  dditpointer,
                                  pointer to descr_type);
4.14.45.7        if  not found
                 7* descr_type = NOTFOUND */
4.14.45.8        then
4.14.45.9          begin
4.14.45.10         dditpointer := null;
4.14.45.11         perform ATM$INSERT(type-C_attr_names[index],
                                      C,
                                      dditpointer);
4.14.45.12       end if;
4.14.45.13     end while;

4.14.45.14 *end_proc;


9.10.2.1  proc  GET_RANDOM_BACKEND_START      /* GETRAND  (DBL1411) */
                              (input: number of backends,
                               output: random_index_to_backends);

          /* Randomly select a backend at which to begin distributing */
          /* data from the current cluster.  Input is a pointer to an */
          /* argument list which contains the number of backends in   */
          /* the system.  Output is a random number generated within  */
          /* the range of 1 to number of backends.                    */

9.10.2.2      scalar  random_index_to_backends;
```

```
9.10.2.3        generate random_index_to_backends
                    within the range 1 to number_of_backends;

9.10.2.4    end proc;


9.10.6.1    proc  DISTRIBUTE_RECORDS              /* DISTREC  (DBL1412) */
**                       (input/output: record,
**                                      next_backend_index,
**                        input: system_info,
**                               cluster_number,
**                        output: capability_remaining);

            /* Physically distribute the data over the multiple backends  */
            /* according to the track-splitting-with-random-placement     */
            /* strategy.                                                  */
**          /* Input is the first record of a cluster, a randomly gener-  */
**          /* ated index into the list of backend addresses, some system */
**          /* information, including the list of backend addresses, and  */
**          /* the cluster number.  Output is the capability remaining in */
**          /* the track at which records added to this cluster are to be */
**          /* stored.  This capacity, together with the index and the    */
**          /* cluster number, will be used to update the CINBT.  Note    */
**          /* also that records are read at this level, so that when the */
**          /* procedure terminates, the variable record will contain the */
**          /* first record of the next cluster.                          */
            /* Records have the form:                                     */
            /*  record=(descr_count, descriptor_ids, database_record).    */
            /* System_info has the form:                                  */
            /*  system_info=(number_of_backends,                          */
            /*               backend_addresses                            */
            /*               track_capacity;                              */
            /* Cluster number is a character string.                      */

9.10.6.2        scalar  capacity_remaining,  /* Capacity remaining on the */
                                             /* track of the backend at   */
                                             /* which the next record of  */
                                             /* this cluster is to be     */
                                             /* stored.                   */
                        next_backend_index,/* Index to backend addresses*/
                        prev_descr_count,    /* Count of descriptor_ids   */
                                             /* from the previous record. */
                        new_cluster;         /* TRUE or FALSE.            */

9.10.6.3        record = (descr_count,       /* Number of descriptor_ids  */
                                             /* in the list following.    */
                         descriptor_ids,     /* List of descriptor ids    */
                                             /* from which this record may*/
                                             /* be derived.               */
                         database_record);   /* Record in format required */
                                             /* for storage.              */

9.10.6.4        array  prev_descr_ids;       /* Descriptor ids from the   */
                                             /* previous record.          */

9.10.6.5        array  full_track;           /* Array in which to accumulate*/
                                             /* a full track of records.    */

9.10.6.6**      capacity_remaining := system_info.track_capacity;

9.10.6.7        while  more records in cluster do
9.10.6.8        begin

                    /* Accumulate a full track of data before distributing   */
                    /* data to the next backend.  When a track is distributed*/
                    /* increment the next_backend_index to point to the next */
                    /* backend address and reset the capacity_remaining.     */
```

```
9.10.6.9            if capacity_remaining >= size(record)
9.10.6.10           then

9.10.6.11               begin
9.10.6.12               add record to full_track array;
9.10.6.13               capacity_remaining :=
                               capacity_remaining - size(record);
9.10.6.14               end

9.10.6.15           else

9.10.6.16               begin
9.10.6.17               distribute cluster_number, full_track to
                               backend at
                               system_info.backend_address
                                                 [next_backend_index];

9.10.6.18               next_backend_index =
                          (next_backend_index + 1)
                                   mod (number_of_backends;

9.10.6.19                capacity_remaining :=
                                system_info.track_capacity;
9.10.6.20           end if;

                    /* Save descriptor count and list of descriptor ids from  */
                    /* the current record for comparison with the next to     */
                    /* detect cluster change.                                 */

9.10.6.21           prev_descr_count := record.descr_count;
9.10.6.22           prev_descr_ids := record.descr_ids;

                    /* Read the next record */

9.10.6.23           read a record from file of sorted records;

9.10.6.24           perform CHECK_FOR_NEW_CLUSTER
                                              (record,
                                              prev_descr_count,
                                              prev_descr_ids,
                                              new_cluster);

9.10.6.25      end while;


9.10.6.26      if  full_track array is not empty
9.10.6.27      then
9.10.6.28           distribute cluster_number, full_track to backend
                           at system_info.backend_address
                                                 [next_backend_index];
9.10.6.29      end if;

9.10.6.30     end proc;
```

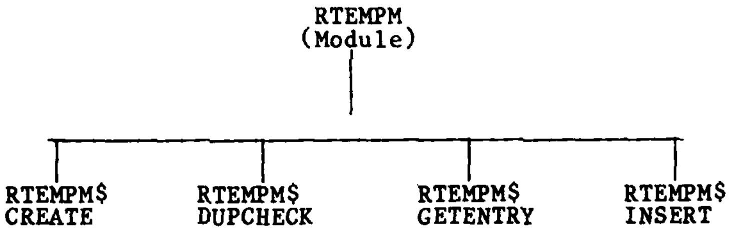
Fifth Level Specification for Database Load

```
9.10.6.25.1   proc CHECK_FOR_NEW_CLUSTER      /* NEWCLUST  (DBL14121) */
                  (input: record,
                          prev_descr_count,
                          prev_descr_ids,
**                 output:  new_cluster);

              /* Check the list of descriptor_ids from the current   */
              /* record against the prev_descr_ids list from the     */
              /* previous record.  If the lists are different lengths */
              /* a new cluster is indicated.  If the lists are the    */
              /* same length, compare them item by item to determine */
              /* whether a new cluster is indicated.                  */
              /* Records have the form:                               */
              /* record=(descr_count, descriptor_ids, database_record)*/

9.10.6.25.2        scalar  index,      /* Index to both lists of ids. */
**                         new_cluster; /* Indicator, TRUE or false.  */


9.10.6.25.3        new_cluster := false;
9.10.6.25.4        if  record.descr_count not =
                                        prev_descr_count
9.10.6.25.5        then
9.10.6.25.6**         new_cluster := true;
9.10.6.25.7        else
9.10.6.25.8           begin
9.10.6.25.9           index := 1;     /* Set index.  */
9.10.6.25.10**        new_cluster := false;
9.10.6.25.11          while index <= record.descr_count do
9.10.6.25.12             begin

9.10.6.25.13             if record.descriptor_ids[index] not =
                                        prev_descr_ids[index]
9.10.6.25.14             then
9.10.6.25.15**              new_cluster := true;
9.10.6.25.16             else
9.10.6.25.17                index := index + 1;

9.10.6.25.18          end while;


9.10.6.25.19      end if

9.10.6.25.20  endproc;
```

C.2 Part II - Record Template Module

```
/*        (1) Part II - Record Template Module          */
/*        (2) Design:    Record Template Module         */
/*        (3) Designer:  P. R. Strawser                 */
/*        (4) Date:      August 25, 1981                */
/*        (5) Modified:                                 */
/*                                                      */
/*        (6) Purpose:                                  */
/*            The record template module provides ser-  */
/*            vices for  record template data structures.*/
/*            A record template data structure is a     */
/*            tabular collection of information about the*/
/*            records of one file, where all records are */
/*            assumed to have the same format.  Each     */
/*            template is identified by record type, and */
/*            contains a count of entries and an entry   */
/*            for each field in the record. Each entry   */
/*            contains field name, data type, length in- */
/*            formation, and an indication of whether the*/
/*            field might be a descriptor.               */
/*                                                      */
/*        (7) Output Data:                              */
/*            A record template for a given file.       */
```

(8) Procedure Structure for RTEMPM



(10) Program Specifications

```
module RTEMPM

programs  CREATE, DUPCHECK, GETENTRY, INSERT;

data structures  record template;

end module


proc  RTEMPM_CREATE(input: record_type,
                    output: rectemppointer);

     /* Name a record template structure with the name record_type   */
     /* and initialize count of entries to zero.                      */
     /* A record template has the structure:                          */
     /*  record_template = (count, entry[no_entries]);                */
     /* An entry in the record template has the structure:            */
     /*  entry = (attr_name, data_type, format, len1, len2,           */
     /*           descr_ind);                                         */

     scalar  rectemppointer;  /* Pointer to record template structure. */


     Allocate a record template data structure with the name
          record_type;

     rectemppointer := pointer to allocated data structure;

     rectemppointer.count := 0;

end proc;


proc  RTEMPM_DUPCHECK(input: rectemppointer,
                             attr_name,
                      output: duplicate);

     /* Check to see whether there is already an entry in this record */
     /* template with an attribute name equal to the input attribute  */
     /* name.  Input is a pointer to the record template and an attri-*/
     /* bute name.  Output is an indicator with a true or false value.*/
     /* A record template has the structure:                          */
     /*  record_template = (count, entry[no_entries]);                */
     /* An entry in the record template has the structure:            */
     /*  entry = (attr_name, data_type, format, len1, len2,           */
     /*           descr_ind);                                         */

     scalar  duplicate,  /* Indicator with TRUE or FALSE value. */
             counter;    /* Local variable.                     */


     counter := 1;
     duplicate := false;

     while  counter is less than or equal to rectemppointer.count
                    & duplicate is false do;
        if attr_name = rectemppointer.entry[counter].attr_name
           then  duplicate := true;
     end while;

end proc;
```

```
proc  RTEMPM_GETENTRY(input: rectemppointer,
                             field_number,
                     output: rectemp_entry);

      /* Get the entry indicated by field_number from the record template */
      /* pointed to by rectemppointer, and return the information to the   */
      /* calling procedure.                                                */
      /* A record template has the structure:                              */
      /*  record_template = (count, entry[no_entries]);                    */
      /* An entry in the record template has the structure:                */
      /*  entry = (attr_name, data_type, format, len1, len2,               */
      /*            descr_ind);                                            */

      /* A record template entry.  */
      rectemp_entry = (attr_name, value_data_type, value_format,
                       value_char1, value_char2, descr_ind);

      if  field_number greater than rectemppointer.count
      then
          rectemp_entry := null;
      else
          rectemp_entry := rectemppointer.entry[field_number];
      end if;

end proc;


proc  RTEMPM_INSERT(input: rectemppointer,
                           rectemp_entry,
                   output: successful);

      /* Insert an entry in the next available slot of the record template */
      /* pointed to by rectemppointer.  Output is an indicator indicating  */
      /* success or failure of the operation.                              */
      /* A record template has the structure:                              */
      /*  record_template = (count, entry[no_entries]);                    */
      /* An entry in the record template has the structure:                */
      /*  entry = (attr_name, data_type, format, len1, len2,               */
      /*            descr_ind);                                            */

      scalar  successful;    /* Indicator with TRUE or FALSE value.  */


      successful := true;

      rectemppointer.count := rectemppointer.count + 1;

      if  rectemppointer.count > maximum fields per record
      then
          successful := false;
      else
          rectemppointer.entry[rectemppointer.count] = rectemp_entry;
      end if;

end proc;
```

APPENDIX D

THE SSL SPECIFICATION FOR DIRECTORY MANAGEMENT

The system specification for directory management is given in this appendix. The specification consists of five parts: the top level of directory management, one service abstraction, and three data abstractions.

In Part I, the top level of directory management is specified. In Part II, the service abstraction employed in directory management is specified. This abstraction, known as directory interface, accepts the output of descriptor search and produces the input for cluster search. The data abstractions for attribute table, descriptor-to-descriptor-id table, and cluster-definition table are specified in Parts III, IV, and V, respectively.

D.1 Part I - The Top Level of Directory Management

```
/*   (1) Part I     : The Top Level of Directory Management          */
/*   (2) Design     : DIRECTORY_MAN                                  */
/*   (3) Designers  : T.M. Ozsu, A. Orooji                           */
/*   (4) Date       : July 28, 1981                                  */
/*   (5) Modified   : Aug. 4, 1981                                   */
/*                    Sept. 11, 1981                                 */
/*   (6) Purpose    :                                                */
/*      This is the directory management subsystem. The  inputs  are a */
/* pointer to a table which contains either the keywords in a record or */
/* the predicates in a query,  either the number of keywords in  the   */
/* record  or  the number of  predicates in the query,  and a schedule */
/* number that is  used  in  determining  the range of  keywords  or   */
/* predicates this backend is supposed to process. The output is either */
/* a cluster id (request type=insert)  or  a set of addresses (request */
/* type=non-insert).                                                  */
```

(8) Procedure Hierarchy for DIRECTORY_MAN

```
                          DIRECTORY_MAN
                                |
      +----------------+--------+----------+----------------------+
      |                |                   |                      |
DIRECTORY_MAN$  DIRECTORY_MAN$      DIRECTORY_MAN$         DIRECTORY_MAN
INS_DESC_SR     NINS_DESC_SR        INS_CLUS_GR            NINS_ADDR_GR
      |               |                  |                           |
      +-------+-------+                  |                           |
              |                          |                           |
     +--------+-------+           +------+-------+                   |
     |        |       |           |              |                   |
  DIRINT$ DIRINT$  DIRINT$      DIRINT$        CDTM$                 |
  CREATE  DEFPRED  BROADCAST    GET_ALL_DESC   FIND_SINGLE_CLUS      |
                                                                     |
                  +------------+-------------+-----------------------+
                  |            |             |
               DIRINT$      DIRINT$        CDTM$
               NO_DESC_GR   NEXT_DESC_GR   FIND_ADDRESS
```

(9) Data Structures

```
/*    The data structure definitions are included in the program */
/* specifications.                                                */
```

(10) Program Specifications

```
1. subsystem DIRECTORY_MAN(input: inptr, number, schedule_no,
                                  output:{cluster_id,addresses});

2.    set addresses;

3.    Find the Attribute Table of the current database, call it AT;
4.    if request type is INSERT
5.       then begin /* inptr=recordptr; number=no. of keywords */
6.           perform INS_DESC_SR(inptr, number, schedule_no, AT);
                /* Do the descriptor search for the keywords in the record */
7.           perform INS_CLUS_GR(inptr, cluster_id);
                /* find the cluster the record belongs to */
8.           return(cluster_id);
9.           end begin
10.      else begin
                /*non-insert; inptr=queryptr; number=no. of predicates */
11.          perform NINS_DESC_SR(inptr, number, schedule_no, AT);
                /* Do the descriptor search phase for the predicates in */
                /* the query                                             */
12.          perform NINS_ADDR_GR(inptr, addresses);
                /* find the addresses of the records in clusters which  */
                /* may satisfy the query                                */
13.          return(addresses);
14.     end if
15. end subsystem;


6.1 proc INS_DESC_SR(input : record_ptr, no_keywords, schedule_no, AT);
    /* This procedure handles the insert cases. Given a record, the number */
    /* of keywords in the record, the schedule number  and  the Attribute  */
    /* Table, it computes the range of keywords it is supposed  to handle  */
    /* and works on the keywords in that range.                            */

6.2   type := 'insert';
```

```
6.3   calculate the range of keywords to work on;
6.4   conjunc_no := 1;
6.5   keyword_no := starting keyword number;
6.6   perform DIRINT$CREATE(request_id); /* create a new RDIT table */
6.7   while there are keywords in range do
6.8      begin
6.9          loc_parameter := (conjunc_no) || (keyword_no);
               /* location parameter consists of conjunction number      */
               /* concatenated with keyword number within that           */
               /* conjunction. In insert cases, cunjunction number is 1  */
6.10         pick next keyword;
6.11         form an equality predicate;
6.12         perform DIRINT$DEFPRED(type,loc_parameter,predicate,AT);
               /* Find the descriptors that satisfy the predicate */
6.13         keyword_no := keyword_no + 1;
6.14  end while
6.15  perform DIRINT$BROADCAST;
        /* Broadcast the descriptor ids to all the other backends */
6.16 end proc;


11.1 proc NINS_DESC_SR(input : query_ptr, no_predicates, schedule_no, AT);
     /* This procedure handles non-insert cases.  Given the query, the    */
     /* total number of predicates in the query, the schedule number and  */
     /* the AT, it computes  the range of predicates  it is supposed to   */
     /* handle and works on the predicates in that range.  We recall that */
     /* each backend handles 1/n of the predicates, where n is number of  */
     /* backends.                                                         */

11.2   type := 'non-insert';
11.3   compute the range of predicates to be worked on;
11.4   conjunct_no := first conjunction number in the range;
11.5   predicate_no := starting predicate number in the range;
11.6   perform DIRINT$CREATE(request_id); /* create a new RDIT table */
11.7   while there are conjunctions in the range do
11.8       begin
11.9           pick next conjunction;
11.10          do begin /* do for each predicate */
11.11              loc_parameter := (conjunct_no) || (predicate_no);
11.12              perform DIRINT$DEFPRED(type,loc_parameter,predicate,AT);
11.13              predicate_no := predicate_no + 1;
11.14          until(end of predicates in this conjunction) or
                                 (end of predicates in the range);
11.15          conjunct_no := conjunct_no + 1;
11.16  end while;
11.17  perform DIRINT$BROADCAST
         /* Broadcast the descriptor ids to all the other backends  */
11.18 end proc;


7.1 proc INS_CLUS_GR(input : recordptr, output : cluster_id);
     /* This procedure finds  the cluster to which  the record  being  */
     /* inserted belongs. If the descriptors of the record  define a   */
     /* new cluster, it signals this to the controller.                */

7.2   list descriptor_id_group; /* used internally for keeping */
                                /* descriptor-id group         */

7.3   wait until RDIT tables are obtained from all backends;
7.4   join all these RDIT tables into one RDIT table;
7.5   perform DIRINT$GET_ALL_DESC(descriptor_id_group);
        /* Get the descriptor-id group for the record being inserted  */
7.6   perform CDTM$FIND_SINGLE_CLUS(descriptor_id_group, cluster_id);
        /* Find the cluster that the record being inserted belongs to */
7.7   return(cluster_id);
        /* If cluster is found, its id is returned. Otherwise a null  */
        /* value is returned.                                         */
7.8 end proc;
```

```
12.1 proc NINS_ADDR_GR(input : queryptr, output : address_list);
        /* This procedure finds the addresses of the records in this */
        /* backend that may satisfy the query.                       */

12.2    scalar conjunct_no, no_group, index : integer;
12.3    list addresses;
12.4    list descriptor_id_group;
12.5    list cluster_nos;

12.6    wait until RDIT tables are obtained from all backends;
12.7    join all these RDIT tables into one RDIT table;
12.8    conjunct_no := 1;
12.9    while there are conjunctions in the query do
12.10       begin
12.11           perform DIRINT$NO_DESC_GR(conjunct_no, no_group);
                  /* find the number of descriptor-id groups for this */
                  /* conjunction.                                     */
12.12           for index from 1 to no_group by 1 do
12.13               begin
12.14                   perform DIRINT$NEXT_DESC_GR(conjunct_no,
                                                descriptor_id_group);
                          /* Get the next descriptor-id group.  */
12.15                   perform CDTM$FIND_ADDRESS(descriptor_id_group,
                                               addresses);
                          /* Find the addresses of the records. */
12.16                   address_list = address_list + addresses;
                          /* Add the addresses found to the address list; */
                          /* caution: duplicates are eliminated.          */
12.17           end for;
12.18           conjunct_no := conjunct_no + 1;
12.19   end while;
12.20   return(address_list);
12.21 end proc;
```

D.2 Part II - The Service Abstraction (DIRINT)

```
/*   (1) Part II    : The Service Abstraction                      */
/*   (2) Design     : DIRINT                                       */
/*   (3) Designers  : T.M. Ozsu,  A. Orooji                        */
/*   (4) Date       : Aug. 4, 1981                                 */
/*   (5) Modified   : Sept. 11, 1981                               */
/*   (6) Purpose    :                                              */
/*     This is the service abstraction employed in directory      */
/* management. This abstraction, known as directory interface,    */
/* accepts the output of descriptor search and produces the       */
/* input for cluster search.                                      */
```

(8) Procedure Hierarchy for DIRINT

```
                                   DIRINT
                                     |
   +--------+---------+-------+---+------------+-----------+
   |        |         |           |            |           |
DIRINT$  DIRINT$   DIRINT$    DIRINT$      DIRINT$     DIRINT$
CREATE   DEFPRED   BROADCAST  GET_ALL_DESC NO_DESC_GR  NEXT_DESC_GR
            |
  +--------+--------+--------+
  |        |        |        |
ATM$    DDITM$   DDITM$   DDITM$
FIND    CDERIVE  DERIVE   INSERT
```

(9) Data Structures

```
/*    The data structure definitions are included in the program */
/* specifications.                                              */
```

(10) Program Specification

```
mod DIRINT
    programs DEFPRED, GET_ALL_DESC, NO_DESC_GR, NEXT_DESC_GR, BROADCAST
    datasets request_descriptor_id_table (RDIT)
         /* A table of (loc_parameter,descriptor id) pairs for */
         /* the present request                                */
end mod


6.12.1 proc DEFPRED(input : type, loc_parameter, predicate, AT);
         /* This procedure finds all the descriptors that satisfy  */
         /* a predicate.                                           */

6.12.2   list desc_ids;  /* list of descriptor ids satisfying */
                         /* the predicate                     */

6.12.3   perform ATM$FIND(AT, attribute, dditptr,descriptor_type);
            /* Find the pointer to DDIT entry for the given attribute */
6.12.4   if search successful
6.12.5      then begin
6.12.6         if (type = 'insert') and (descriptor_type = 'C')
6.12.7            then begin
6.12.8               perform DDITM$CDERIVE(predicate,dditptr,desc_ids);
6.12.9               if keyword not derivable
6.12.10                 then begin  /* a new type-C descriptor */
6.12.11                    new_desc_id := a new descriptor id;
                                /* Give this descriptor a new id  */
                                /* and insert it into DDIT        */
6.12.12                    perform DDITM$INSERT(descriptor,
                                          new_desc_id, dditptr2);
                                /* insert the new descriptor into DDIT */
6.12.13                    value(RDIT,loc_parameter) := new_desc_id;
                                /* Insert the new descriptor id into RDIT */
6.12.14                    end begin
6.12.15                 else
6.12.16                    value(RDIT,loc_parameter) := desc_ids;
6.12.17                 end if
6.12.18              end begin
6.12.19           else begin
6.12.20              perform DDITM$DERIVE(predicate, dditptr,desc_ids);
                         /* Find those descriptors from which this   */
                         /* predicate is derivable and put their ids */
                         /* into the desc_ids list                   */
6.12.21              value(RDIT,loc_parameter) := desc_ids;
                         /* add a new pair for each descriptor id in */
                         /* desc_ids list                            */
6.12.22           end if
6.12.23  end if
6.12.24 end proc;


7.5.1 proc GET_ALL_DESC(output : descriptor_id_group);
        /* This procedure gets  the descriptor ids  of  the descriptors */
        /* from which  the keywords  in a record  have been found to be */
        /* derivable.                                                   */

7.5.2   descriptor_id_group := null;
7.5.3   do  /* collect all descriptor ids */
7.5.4      descriptor_id_group := descriptor_id_group +
                    descriptor id at current location of RDIT;
7.5.5   until(end of RDIT table);
7.5.6   return(descriptor_id_group);
7.5.7 end proc;
```

```
12.11.1 proc NO_DESC_GR(input : conjunct_no, output : no_group);
           /* This procedure finds  the number of descriptor-id groups for  */
           /* the given conjunction.  Furthermore, it initializes  certain  */
           /* arrays and counters that will be used by NEXT_DESC_GR         */

12.11.2    array counter, desc_per_pred : integer;  /* global arrays */
12.11.3    scalar pred_no, index : integer;

12.11.4    find the beginning of predicates for conjunct_no in RDIT;
12.11.5    pred_no := 1;
12.11.6    no_group := 1;
12.11.7    index := 1;
12.11.8    do begin   /* initialize desc_per_pred array */
12.11.9          desc_per_pred(index) := 0;
12.11.10         index := index + 1;
12.11.11   until(end of desc_per_pred array);
12.11.12   do begin  /* Calculate number of descriptors for all */
                     /* predicates in the given conjunction.    */
12.11.13        loc_param := conjunct_no || pred_no;
12.11.14        do    /* calculate no. of descriptors for one predicate */
12.11.15             pick next RDIT entry;
12.11.16             desc_per_pred(pred_no) := desc_per_pred(pred_no) + 1;
12.11.17        until(loc_param ~= location parameter in RDIT) or
                                                   (end of RDIT);
12.11.18        no_group := no_group * desc_per_pred(pred_no);
                       /* keep a running total of number of */
                       /* descriptor-id groups              */
12.11.19        pred_no := pred_no + 1;
12.11.20   until(end of conjunction);
12.11.21   desc_per_pred(0) := pred_no - 1;
               /* keep the total no. of predicates in conjunction */
12.11.22   index := 1;
12.11.23   do begin /* set counter array to 1; to be used */
                    /* in NEXT_DESC_GR                    */
12.11.24        counter(index) := 1;
12.11.25        index := index + 1;
12.11.26   until(end of counter array);
12.11.27   return(no_group);
12.11.28 end proc;


12.14.1 proc NEXT_DESC_GR(input : conjunc_no,
                               output : descriptor_id_group);
          /* This procedure generates the next descriptor-id group that */
          /* satisfies the predicates in the conjunction identified by  */
          /* conjunc_no.                                                */

12.14.2   array desc_per_pred, counter : integer;  /* global arrays */
12.14.3   scalar index, eff_index : integer;

12.14.4   find the beginning of predicates for conjunc_no in RDIT;
12.14.5   eff_index := beginning position;
          /* The for loop finds the next descriptor-id group */
12.14.6   descriptor_id_group := null;
12.14.7   for index from 1 to no. of predicates by 1 do
12.14.8      begin
12.14.9          descriptor_id_group := descriptor_id_group +
                      descriptor id at RDIT(eff_index+counter(index));
12.14.10         eff_index := eff_index + desc_per_pred(index);
12.14.11     end for;
           /* In the remainder, the counter array is updated for the   */
           /* next invocation                                          */
12.14.12   index := no. of predicates;
12.14.13   counter(index) := counter(index) + 1;
             /* indicate  that the next  descriptor id  for the last   */
             /* predicate will be picked up next time                  */
12.14.14   while counter(index) > desc_per_pred(index) do
               /* If the last descriptor id  for  that predicate  is */
               /* already  picked up,  indicate that  the  next      */
               /* descriptor id for the predicate  immediately  prior */
               /* to this one will be picked up next time,  together  */
```

```
                 /* with  the first descriptor id  of  this predicate.  */
                 /* This is done  by  setting  the  counter  entry      */
                 /* corresponding  to  present  predicate  to  1  and   */
                 /* incrementing the counter entry corresponding to the */
                 /* immediately previous one. Keep doing this until no  */
                 /* more adjustments are necessary.                     */
12.14.15        begin
12.14.16            counter(index) := 1;
                         /* next time the first descriptor id for this */
                         /* predicate will be picked up                */
12.14.17            index := index - 1;  /* look at the previous predicate */
12.14.18            if index = 0 /* if all the descriptor-id groups have */
                                /* been picked                           */
12.14.19                then exit  /* leave the loop */
12.14.20                else counter(index) := counter(index) + 1;
                             /* else increment count for the previous    */
                             /* predicate                                */
12.14.21                end if
12.14.22        end while
12.14.23    return(descriptor_id_group);
12.14.24 end proc;
```

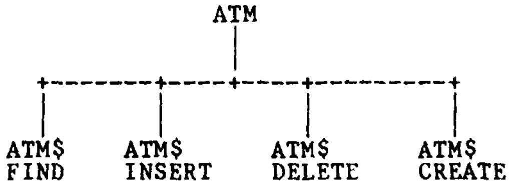
```
6.15.1 proc BROADCAST;
         /* This procedure broadcasts the RDIT to all the other backends */
6.15.2   broadcast RDIT;
6.15.3 end proc;
```

```
6.6.1 proc CREATE(input: request_id);
        /* This procedure creates an occurrence of RDIT table for the */
        /* given request.                                             */
6.6.2 end proc;
```

D.3 Part III - The Data Abstraction for Attribute Table

```
/*   (1) Part III  : The Data Abstraction for Attribute Table          */
/*   (2) Design    : ATM                                               */
/*   (3) Designers : T.M. Ozsu,  A. Orooji                             */
/*   (4) Date      : July 28, 1981                                     */
/*   (5) Modified  : Aug. 4, 1981                                      */
/*                   Sept. 11, 1981                                    */
/*   (6) Purpose   :                                                   */
/*      This is the data abstraction for  attribute table.  Operations */
/* on attribute table are done via the procedures in this abstraction. */
```

(8) Procedure Hierarchy for ATM



(9) Data Structures

```
/*   The data structure definitions are included in the program */
/* specifications.                                              */
```

(10) Program Specifications

```
mod ATM;
   programs FIND, INSERT, DELETE, CREATE
   datasets AT /* attribute table */
end mod;


6.12.3.1 proc FIND(input : AT, attribute, output : dditptr,type);
           /* This procedure finds the location of the attribute in AT. */
           /* It returns the pointer to the  DDIT  for  that  attribute */
           /* and the type of descriptors specified on the attribute.   */

6.12.3.2   find the matching attribute;
6.12.3.3   dditptr := pointer at that position;
6.12.3.4   type := type of descriptors defined on that attribute;
6.12.3.5   return(dditptr, type);
6.12.3.6 end proc;


1. proc INSERT(input : AT, attribute, dditptr);
    /* This procedure inserts an (attribute, pointer) pair into AT. */
2.  search for the position where attribute fits;
3.  insert new (attribute, dditptr) pair;
4. end proc;


1. proc DELETE(input : AT, attribute);
    /* This procedure deletes an (attribute, pointer) pair from AT. */
2.  find the matching attribute;
3.  delete the entry at that position;
4. end proc;


1. proc CREATE;
    /* This procedure creates a new instance of the attribute table */
    /* and returns a pointer to it.                                 */
2.  create a new instance of the attribute table;
3.  insert the database name together with the pointer to the
           new AT into the index table for AT's;
4. end proc;


1. proc UPDATE(input : AT, attribute, dditptr);
    /* This procedure updates the dditptr of the given attribute to */
    /* the new dditptr given as input.                              */
2.  find the attribute in AT;
3.  replace the dditptr for the attribute with the new one;
4. end proc;
```

D.4 Part IV - The Data Abstraction for Descriptor-to-Descriptor-Id Table

```
/*   (1) Part IV   : The Data Abstraction for DDIT                       */
/*   (2) Design    : DDITM                                               */
/*   (3) Designers : T.M. Ozsu, A. Orooji                                */
/*   (4) Date      : July 28, 1981                                       */
/*   (5) Modified  : Aug. 4, 1981                                        */
/*                   Sept. 11, 1981                                      */
/*   (6) Purpose   :                                                     */
/*      This is the data abstraction for DDIT.  Operations  on  DDIT are */
/* done via procedures in this abstraction.                              */
```

(8) Procedure Hierarchy for DDITM

```
                              DDITM
                                |
      +----------+--------------+----------+----------+
      |          |              |          |          |
   DDITM$     DDITM$         DDITM$     DDITM$     DDITM$
   CDERIVE    DERIVE         INSERT     DUPCHECK   CREATE
```

(9) Data Structures

```
/*  The data structure definitions are included in the program */
/* specifications.                                             */
```

(10) Program Specifications

```
mod DDITM
   programs DERIVE, CDERIVE, INSERT, DUPCHECK, CREATE
   datasets DDIT    /* Descriptor-to-descriptor-id table */
end mod;


6.12.20.1 proc DERIVE(input : predicate, dditptr, output : desc_ids);
             /* This procedure finds out the ids of all the  descriptors  */
             /* from which the predicate can be derived and returns these */
             /* ids in desc_ids.  This routine is used for all the cases  */
             /* except  when  the  request is insert and attribute of the */
             /* keyword is used in type-C descriptors.                    */

6.12.20.2   desc_ids := null;
6.12.20.3   do begin
6.12.20.4      if predicate derivable from descriptor at DDIT(dditptr)
6.12.20.5           then
6.12.20.6               desc_ids := desc_ids +
                                    descriptor id at DDIT(dditptr);
6.12.20.7      end if;
6.12.20.8      dditptr:= next entry position in DDIT;
6.12.20.9   until (descriptors on the same attribute as predicate´s finishes)
6.12.20.10  return(desc_ids);
6.12.20.11 end proc;


6.12.8.1 proc CDERIVE(input : predicate, dditptr, output : desc_id);
            /* This procedure finds out the id of the descriptor  from */
            /* which the predicate can be derived and returns this id. */
            /* This routine  is used  only when the request is  insert */
            /* and attribute  of  the  keyword  is used  in  type-C    */
            /* descriptors.                                            */

6.12.8.2   do begin
6.12.8.3      if predicate derivable from descriptor at DDIT(dditptr)
6.12.8.4         then begin
6.12.8.5                 desc_id := descriptor id at DDIT(dditptr);
6.12.8.6                 return;
6.12.8.7      end if
6.12.8.8      dditptr := next entry position in DDIT;
6.12.8.9   until (descriptors on the same attribute as predicates´s finishes)
6.12.8.10  return(´not derivable´);
6.12.8.11 end proc;
```

```
proc DUPCHECK(input : descriptor, dditptr, output : answer);
    /* Given a  descriptor, this procedure checks to make sure that */
    /* its  range  does not  overlap  the ranges of  other  already */
    /* defined descriptors in DDIT.                                 */

    answer := 'no';
    do begin
        if descriptor range overlaps the range of that
                                   pointed at by dditptr
            then begin
                answer := 'yes';
                return;
        end if;
        dditptr := next descriptor in DDIT defined on the same attribute;
    until (no more descriptors on the same attribute);
end proc;
```

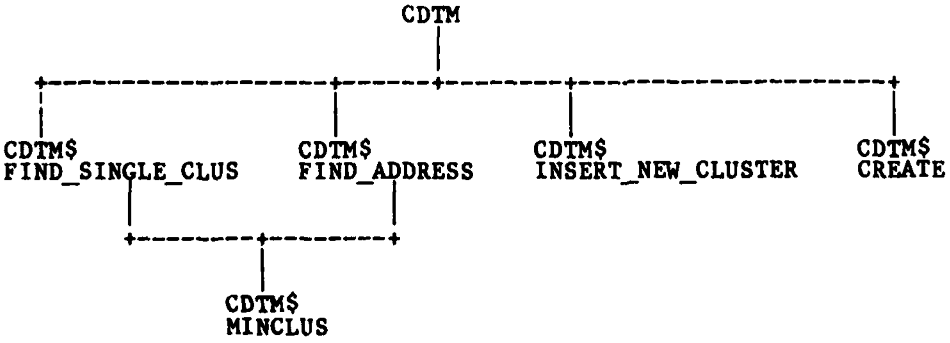
```
6.12.12.1 proc INSERT(input : descriptor, desc_id, output : dditptr);
              /* This procedure inserts a descriptor and its id into DDIT. */
6.12.12.2   find the place for the descriptor;
6.12.12.3   insert the descriptor;
6.12.12.4 end proc;
```

```
proc CREATE;
    /* This procedure creates a new instance of  the descriptor-to-  */
    /* descriptor-id table.                                          */
    create a new instance of DDIT;
    insert the database name together with the pointer to the
            new DDIT into the index table for DDIT's;
end proc
```

D.5 Part V - The Data Abstraction for Cluster-Definition Table

```
/*   (1) Part V     : The Data Abstraction for CDT                     */
/*   (2) Design     : CDTM                                             */
/*   (3) Designers  : T.M. Ozsu,  A. Orooji,  Z. Shi                   */
/*   (4) Date       : July 31, 1981                                    */
/*   (5) Modified   : Aug. 7, 1981                                     */
/*                    Sept. 11, 1981                                   */
/*   (6) Purpose    :                                                  */
/*      This is the data abstraction for cluster-definition table.     */
/* Operations  on  CDT are  done  via  the  procedures  in  this       */
/* abstraction.                                                        */
```

(8) Procedure Hierarchy for CDTM



(9) Data Structures

```
/*  The data structure definitions are included in the program */
/* specifications.                                              */
```

(10) Program Specifications

```
mod CDTM ;
   programs FIND_SINGLE_CLUS, FIND_ADDRESS,
               INSERT_NEW_CLUSTER, CREATE;
   datasets ECDT, descriptor table (DT),
               descriptor-to-cluster map (DTCM)
end mod;


7.6.1 proc FIND_SINGLE_CLUS(input : desc_id_group, output : cluster_id);
         /* This procedure finds the cluster whose descriptor-id set */
         /* matches desc_id_group                                     */

7.6.2    scalar stop : boolean;
7.6.3    scalar index : integer;
7.6.4    scalar mindesc : character;

7.6.5    cluster_id := null;
7.6.6    perform MINCLUS(desc_id_group, mindesc);
            /* Among the descriptor ids in desc_id_group, find the */
            /* id whose descriptor participates in defining the    */
            /* smallest number of clusters                         */
7.6.7    do begin  /* this loop looks at each cluster whose */
                   /* descriptor-id set contains mindesc    */
7.6.8         pick next entry in DTCM for this descriptor;
7.6.9         pick the entry in ECDT pointed at by cdtptr in
                   the current DTCM entry;
7.6.10        if ECDT(cdtptr).no_desc = no. of descriptors in desc_id_group
7.6.11             then begin  /* the descriptor-id set for this cluster */
                               /* may match since it has the same number */
                               /* of descriptors                         */
7.6.12                  index := 1;
7.6.13                  stop := 'false';
7.6.14                  do  /* look at each descriptor id in */
                            /* descriptor-id set             */
7.6.15                      if descriptor id currently pointed at by descptr ~=
                                                  desc_id_group(index)
7.6.16                         then stop := 'true'  /* no match; stop */
7.6.17                         else begin /* match, pick next descriptor id */
                                          /* in each list                   */
7.6.18                              index := index + 1;
7.6.19                              update descptr to point to
                                                next descriptor id;
7.6.20                         end if
7.6.21                  until(end of descriptors in desc_id_group) or (stop);
7.6.22                  if not stop    /* see if there was a match */
7.6.23                      then begin /* there was a match        */
7.6.24                         cluster_id := cluster id at ECDT(cdtptr);
7.6.25                         return;
7.6.26                      end if
7.6.27             end if
7.6.28   until (no more entries in DTCM for this descriptor);
7.6.29 end proc;


12.15.1 proc FIND_ADDRESS(input : desc_id_group, output : addresses);
           /* This  procedure  finds  the addresses  of  the records  in  */
           /* clusters whose descriptor-id set  contain desc_id_group     */

12.15.2    scalar index : integer;
12.15.3    scalar stop :boolean;

12.15.4    addresses := null;
```

```
12.15.5   perform MINCLUS(desc_id_group, mindesc);
             /* Among the descriptor ids in desc_id_group, find the   */
             /* id  whose  descriptor  participates  in  defining the */
             /* smallest number of clusters.                          */
12.15.6   do begin  /* do for all the clusters whose descriptor-id */
                    /*set contain mindesc                          */
12.15.7      pick next entry in DTCM for this descriptor;
12.15.8      pick next entry in ECDT pointed at by cdtptr in
                  the current DTCM entry;
12.15.9      if ECDT(cdtptr).no_desc >= no. of descriptors in desc_id_group
12.15.10         then begin  /* the descriptor-id set for this cluster */
                             /* may contain desc_id_group              */
12.15.11            index := 1;
12.15.12            stop := 'false';
12.15.13            do /* look at each descriptor id in the  */
                       /* descriptor-id set for this cluster */
12.15.14               if descriptor id currently pointed at by descptr >

                                          desc_id_group(index)
12.15.15                  then stop := 'true';
                               /* descriptor-id set does not contain */
                               /* desc_id_group(index)               */
12.15.16                  else if descriptor id pointed at by descptr =
                                          desc_id_group(index)
12.15.17                       then begin
                                  /* match; look at next id in */
                                  /* both lists                */
12.15.18                            index := index + 1;
12.15.19                            update descptr to point to next
                                          descrptor id;
12.15.20                            end begin
12.15.21                       else update descptr to point to next
                                          descriptor id;
                                  /* keep looking in the */
                                  /* descriptor-id set   */
                                  /* for the cluster     */
12.15.22                       end if
12.15.23                  end if
12.15.24            until(end of descriptors in desc_id_group)
                           or (descptr = null) or (stop);
12.15.25            if (index > no. of desc. in desc_id_group)
12.15.26                then  /* the search was successful;     */
                              /* add addresses of the records   */
                              /* in this cluster to the list of */
                              /* those which qualify            */
12.15.27                  addresses := addresses +
                              addresses pointed at by addrptr in
                                    ECDT(cdtptr);
12.15.28            end if
12.15.29         end if
12.15.30  until(no more entries in DTCM for this descriptor);
12.15.31 end proc;


1.  proc MINCLUS(input : desc_id_group, output : mindesc);
      /* Among the descriptor ids in desc_id_group,  this  procedure */
      /* finds the id whose descriptor participates in defining the  */
      /* smallest number of clusters.                                */

2.    scalar min : integer;

3.    find the first descriptor id in desc_id_group in DT (call it cur_desc);
4.    min := DT(cur_desc).no_clus;
5.    mindesc := current descriptor id;
6.    do begin   /* do for all ids in desc_id_group */
7.       find next descriptor id in desc_id_group in DT;
8.       if DT(cur_desc).no_clus < min  /* see if the current one is min */
9.          then begin   /* yes, make the current one min */
10.            min := DT(cur_desc).no_clus;
11.            mindesc := current descriptor id;
12.         end if
13.   until(no more ids in desc_id_group);
14.   return(mindesc);
15. end proc;
```

```
proc INSERT_NEW_CLUSTER(input : desc_id_set, cluster_id,
                                            output : cdtptr);
    /* This procedure inserts a new cluster into ECDT and updates */
    /* all the other tables accordingly.  It returns a pointer to */
    /* this new entry.                                            */

    sort desc_id_set in ascending order of descriptor ids;
    /* update ECDT table */
    create a new ECDT entry (call it new_cdt);
    new_cdt.cluster_id := cluster_id;
    new_cdt.no_desc := no of descriptors in desc_id_set;
    form a linked list of descriptor ids in desc_id_set;
    update new_cdt.descptr to point to the linked list of descriptor ids;
    add new_cdt to the ECDT list;
    /* update DTCM */
    create DTCM entries for all descriptor ids in
                 desc_id_set (call it new_dtcm);
    set cdtptr of all new_dtcm's to point to the new_cdt entry;
    add new_dtcm entries to their respective DTCM lists for each
        descriptor id;
    /* update DT table */
    update cluster counts (no_clus) of DT entries for the
               descriptor ids in desc_id_set;
end proc;
```

```
proc CREATE(output : cdtptr);
   /* This procedure creates a new ECDT and returns a pointer to it */
   create a new instance of ECDT;
   insert the database name together with the pointer to the
          new ECDT into the index table for ECDT's;
   return the pointer to the new ECDT;
end proc;
```

ATE

LMED

-8